情報科学
2009年度版

増原英彦

平成21年10月4日

2009 年度版について

このファイルは東京大学教養学部の科目「情報科学」のために書かれた教科書である。授業の進行に合わせて加筆・修正が加えられたものが、下記の授業ページで随時公開されてゆくので、適宜確認をして下さい。

http://lecture.ecc.u-tokyo.ac.jp/johzu/joho-kagaku/

また、授業の履修者かどうかに関らず、誤り、疑問点、ご意見、ご感想を歓迎します。電子メールにて下記までお知らせ下さい。

masuhara+is+text@lecture.ecc.u-tokyo.ac.jp

はじめに

本書の目標は、情報科学の基本概念や思考方法をプログラミングを通して習得することである。情報科学とは、コンピュータの発明によって拓かれた比較的新しい科学であるが、単にコンピュータの作り方や使い方にとどまらず、量子コンピュータや遺伝子情報処理などのように物理学・化学・生物学などの伝統的な科学の手法や思考方法にも大きな影響を与えるようになっている。本書はそのような情報科学の入門として、アルゴリズム・計算量・数値計算・パターン認識・モデル化などの話題を取りあげている。ただしこれらの話題は、情報科学の中でも入門的かつ代表的な項目から、ごく一部を選んだものであることに注意されたい。

本書ではこれらの話題について、プログラムを動かしながら学習する。初学者でも自分でプログラムを書き、動かすことができるようにするため、必要とするプログラミングの知識をできる限り少なくなるように絞り込み、本書の前半 (第 1 章から第 4 章) で簡単なプログラムを書くための必要最小限の解説をする。後半 (第 5 章から第 9 章) は情報科学の概念や思考方法を学ぶ章である。

本書はプログラミング言語 Ruby を使用する。ただし前述のように必要最小限の機能に絞った紹介するので、プログラミング言語の入門としてはごく基本的な段階にとどまっている。従って例題を Ruby 以外のプログラミング言語に置き換ても本書の内容を学習することも可能だろう。他のプログラミング言語との関係については、本書でも第 10 章で相違点などを説明している。

本書には学習内容を確認するために多くの練習問題が用意されている。特に本文中にある練習問題は、後で必要となる関数を定義しているなど大事なものが配されている。それ以外の練習問題は各章の最後に書かれているが、学習順序としては章末の練習問題も適宜解きながら読み進める方が効果的であろう。

本文中の説明のうち、進んだ話題を取り上げており、順に読まなくとも構わないものには印 (*) がつけられている。練習問題も難しものには同じ印を付けているので、自習の際の参考にしてほしい。

目 次

第I部

必要最小限のプログラミング

第1章　数の計算と関数

数式の計算は、コンピュータの発明以前から人間が行ってきた情報処理の1つであるが、プログラミングと共通する性質を多く持っている。まずは我々には馴染み深い数式の計算をコンピュータに行わせる方法を見ることで、プログラミングの世界への第一歩を踏み出そう。

1.1　コンピュータとの対話

本書では、irb と呼ばれるソフトウェアを用いて実際にプログラムを動かしながら様々な考え方が概念を学んでゆくことになる。irb はプログラミング言語 Ruby を対話的に使用するためのプログラミング環境である。対話的な環境とは、人間が質問や指示を1つ出すごとにコンピュータが即座に処理を行って返答するものなので、あたかも会話をするかのようにプログラミングを進めることができる。そのためプログラミングの学習やソフトウェア開発を試行錯誤しながら行うことに適している。

まずは irb を起動して簡単な数式の計算を行わせる方法を見る。なお、ここでは使用するコンピュータに irb がすでにインストールされていることを前提としている。インストール方法や OS による具体的な起動手順などは以下のページを参照せよ。

http://lecture.ecc.u-tokyo.ac.jp/johzu/joho-kagaku/

多くの OS では、irb を使って計算をさせる場合には次のような手順を踏むことになる:

ターミナルの起動 ⇒ irb の起動 ⇒ 数式の入力

まず最初に「ターミナル」を起動する。このソフトウェアは OS に対する命令をキーボードから入力するためのものである。

起動後はウインドウが開かれ、キーボードから入力できる状態になったことを示す**プロンプト**と呼ばれる文字列が表示される。

```
1 cm12345$ ▉
```

この画面では cm12345$がプロンプトであるが、使っている OS やコンピュータによって異なる文字列が表示される。なお ▉は文字が入力されてゆく位置を示す**カーソル**であり、これの形状も OS やコンピュータによって異なる。

OS ごとに異なるソフトウェアを参照する必要がある場合、本書では Mac OS X 日本語版における名称を用いる。「ターミナル」はオペレーティングシステムによって「コマンド プロンプト」や「Terminal」といった名前が付いている。

次にキーボードから irb Return と入力する。これは「irb という名前のソフトウェアを起動せよ」という OS に対する命令である。irb もまたキーボードから入力して計算を行うソフトウェアなので、今度は irb のプロンプトが表示される。なお、以降では Return を入力することを ↵ で表わす。

キーボードによっては Enter と書かれているかも知れない。

```
1 cm12345$ irb↵
2 irb(main):001:0> 
```

上ではキーボードからの入力に下線を引いてある。「irb(main):001:0>」という部分が irb のプロンプトである。

ここで適当な数式を入力すると、その式が計算され結果が表示される。例えば 1+1 ↵ と入力してみよう。

```
2 irb(main):001:0> 1+1 ↵
3 => 2
4 irb(main):002:0> 
```

上の面画の 3 行目が結果で、=>に続く「2」が計算結果である。次の行に再びプロンプトが表示され、さらに別の式が入力できる状態になっている。他にも式を入力して計算が行われることを確認してみよ。

実行画面やプログラムの行番号は、欄の左側に書かれている。

irb を終了するには、irb のプロンプトが表示されている状態で Control D を入力する。

Control あるいは Ctrl と書かれたキーを押しながら D を入力する。

```
4 irb(main):002:0> Control D
5 cm12345$ 
```

ターミナルから irb を起動した場合はターミナルのプロンプトが表示されるので、再び irb を起動したり、ターミナルを終了させることができる。

1.2　数式の計算

1.2.1　電卓がわり

まずはどのような数式の計算ができるか、例を見てゆこう。

```
1 irb(main):003:0> 7 - 2  # 引き算
2 => 5
3 irb(main):004:0> 7 * 2  # 掛け算
4 => 14
5 irb(main):005:0> 7 / 2  # 割り算の商
6 => 3
7 irb(main):006:0> 7 % 2  # 割り算の余り
8 => 1
```

```
irb(main):007:0> 7 ** 2 # 冪乗
=> 49
```

上の画面で数式の後ろに「# 引き算」などと書かれているが、この部分は説明のために書き加えた部分であり、実際には入力する必要はない。ただし # から後ろは無視されるので、入力しても問題はない。

ここに示した式の意味は説明するまでもないものがほとんどだろうが、いくつか注意が必要である。まず、$\times, \div$ のような記号や x^y のような書き方をキーボードから直接行うことができないため、*, /, %, ** などの記号を使っている。また、5 行目の割り算の結果は 3.5 とはならず、小数点以下を切り捨てた整数商になっている。

もう少し複雑な式の例も見ておこう。長い式を書いたときには、どこから計算するかによって結果が違ってしまうが、括弧を省略した場合は人間が手で計算するときと同様の、「乗除算は加減算よりも優先」などの決まりに従って計算される。

```
irb(main):009:0> 7 - 2 * 3
=> 1
irb(main):010:0> (7 - 2) * 3
=> 15
```

数の計算について一つだけ注意しなければいけないことがある。それは、整数どうしを / によって割ると結果は整数商になるが、Ruby では除数か被除数のどちらか一方でも実数であった場合は、結果も実数になるということである。

```
irb(main):012:0> 7.0 / 2
=> 3.5
irb(main):013:0> 7   / 2.0
=> 3.5
```

つまり Ruby では**整数 1 と実数 1.0 は違う計算結果を与えることがある**ので、特に実数の割り算を行いたい場合には注意せよ。

練習 1.1 (数式の計算) 次の計算を行う Ruby の式と、その結果を書け。

a) 56 と 97 と 33 の和

b) 47 引く 38

c) 35 掛ける 22

d) 34 を 15 で割った商

e) 34 を 15 で割った余り

f) 34 割る 15 (実数で)

g) 56 の 16 乗 (**を使って)

h) 56 の 16 乗 (**を使わずに)

i) 野球選手清原和博の Body Mass Index (BMI). 身長は 188 cm 体重は 104 kg とせよ。また BMI とは体重 (kg) を身長 (m) の自乗で割った値である。

(章末の練習問題: 1.3)

1.2.2　数学関数

三角関数や対数などの数学関数の計算には、**関数**を使った計算をする。

Ruby では**メソッド**と呼ばれているが、本書ではより一般的な名称である関数という言い方を用いる。

Ruby では数学関数を使う場合は `include(Math)` という命令を実行しておく必要がある。

```
19 irb(main):003:0> include(Math)      #数学関数を使う準備
20 => Object
21 irb(main):004:0> sqrt(2)
22 => 1.4142135623731
23 irb(main):005:0> cos(3.141592/3)
24 => 0.50000018867511
```

上の例の 19 行目が、数学関数を使うための準備である。21 行目は $\sqrt{2}$ の計算、23 行目は $\cos(\pi/3)$ の計算である。三角関数に与える角度は 0° から 360° を 0 から 2π に対応させたラジアンによって与えていることに注意せよ。

Ruby で使える代表的な数学関数には、以下のものがある。

関数の使い方	意味	関数の使い方	意味
`sin(x)`	$\sin x$	`exp(x)`	e^x
`cos(x)`	$\cos x$	`log(x)`	$\log x$
`tan(x)`	$\tan x$	`log10(x)`	$\log_{10} x$
`sqrt(x)`	$\sqrt{x}$		

(章末の練習問題: 1.4)

1.3　変数

1.3.1　値に名前をつける

ここまでで見た数式は、例えばBMIを求める式 `104 / (188 / 100.0) ** 2` のようにすべて数値だけを使って表わしていた。このような式は、変数を

使って一般化して「身長 h cm, 体重 w kg の BMI を $\frac{w}{(h/100.0)^2}$ と定める」のように書くことができる。

プログラミング言語でも、**変数**を使って数式を書くことができるので、その方法を見てみよう。変数を使うためには、まず変数を定義する。

```
irb(main):003:0> h=188.0
=> 188.0
irb(main):004:0> w=104.0
=> 104.0
```

上の `h=188.0` のような命令は `h` という名前の変数を定義して、その値を 188 にしている。これを**変数代入**あるいは単に**代入**という。

変数を使った式は次のようになる。

```
irb(main):006:0> w / (h/100.0) ** 2
=> 29.4250792213671
```

このように、数値のかわりに変数名を書くだけである。これを**変数の参照**という。

(章末の練習問題: 1.5 1.6 1.7)

1.3.2　変数を使うわけ *

数式 `104.0 / (188.0/100.0) ** 2` を変数を使って `w / (h/100.0) ** 2` のように書き換えても、やっている演算は変わらない。それにも関らず、変数を使って数式を書き直すことにはどのような理由があるのだろうか。

式の意味が理解しやすくなる

一つには `104.0 / (188.0/100.0) ** 2` という式は、たとえこれが BMI の計算だと知っていたとしても、その中の 104.0 や 188.0 という値が何を表わしているかは即座には分からない。しかし `w / (h/100.0) ** 2` であれば、`w` が体重 (weight) で `h` が身長 (height) であることは名前から容易に想像が付くだろう。つまり、変数を使うと式に現われる値の意味が分かりやすくなるということだ。

さらに Ruby を含むほとんどのプログラミング言語では、`w` のかわりに `weight`, `h` のかわりに `height` などのように、2 文字以上の名前の変数を使うことができる。変数が表わす値の意味をよりはっきりさせる必要がある場合には、`body_weight_in_pound` (ポンドで表わした体重) のように単語を下線記号 (_) でつなげた名前の変数を使うこともできる。この下線記号は、アルファベットと同様の扱いを受けるので、`body_weight_in_pound` で一つの変数になる。

値を変えた計算のやり直しができる

変数を使うと、式の一部の値を変化させた場合の計算、つまり計算のやり直しが簡単にできる。例えば同じ身長で体重が 10 kg 減ったの場合の BMI を求めることを考えよう。変数を使った式の場合、該当する変数 w を定義し直せば、あとは同じ式を計算するだけでよい。

```
7  irb(main):008:0> w=104.0-10
8  => 94.0
9  irb(main):009:0> w / (h/100.0) ** 2      #前と同じ式
10 => 26.5957446808511
```

なお irb では↑, ↓キーを押すと以前に入力した式が画面に現われる。9 行目では、実際には式を入力せず、↑キーを何回か押して 5 行目の式を表示させて計算させることができる。

1.4　関数の定義

1.4.1　BMI を求める関数

色々な人の BMI を計算しなければいけなくなったとしよう。数学であれば身長 height と体重 weight が与えられたときの BMI を

$$\mathrm{BMI}(\mathrm{height}, \mathrm{weight}) \equiv \frac{\mathrm{weight}}{(\mathrm{height}/100)^2} \tag{1.1}$$

と定義する、と言っておけばあとは $\mathrm{BMI}(188, 104)$ とか $\mathrm{BMI}(174, 119\times 0.454)$ のような式を書くことができる。

Ruby では関数を定義することでこれと同じことができる。

```
1  irb(main):003:0> def bmi(height,weight)
2  irb(main):004:1>     weight / (height/100.0) ** 2
3  irb(main):005:1> end
4  => nil
```

上の画面では def と end が関数を定義していることとその範囲を表わしていて、1 行目が式 1.1 中 ≡ の左側、2 行目が右側に対応する。

このようにして定義した関数は数学関数 (第 1.2.2 節) と同様に使うことができる。

```
5  irb(main):007:0> bmi(188.0,104.0)
6  => 29.4250792213671
7  irb(main):008:0> 1.1*bmi(174.0, 119.0 * 0.454)
8  => 19.6289470207425
```

練習 1.2 (関数の定義) 次のような関数を定義せよ。さらに、それを使った計算の例を示せ。

a) 平面上の 2 点 (x, y) と (u, v) の距離を求める `distance(x,y,u,v)`.

b) f フィート i インチをセンチメートルに変換する `feet_to_cm(f,i)`. ただし、1 フィート = 12 インチ = 30.48 cm である。

c) p ポンド o オンスをキログラムに変換する `pound_to_kg(p,o)`. 1 ポンド = 16 オンス = 0.4536 kg である。

(章末の練習問題: 1.8)

1.4.2　関数を使う関数

第 1.4.1 節で定義したような関数は、数学関数と同じように使えるので、別の関数の定義の中でも使うことができる。例えば、身長 f フィート i インチ、体重 p ポンド o オンスに対する BMI を計算する `bmi_yp(f,i,p,o)` を定義することを考えよう。すでに関数 `bmi`, `feet_to_cm` (練習 1.2b), `pound_to_kg` (練習 1.2c) は定義済みなので、これらを使って次のように定義すればよい。

```
1 irb(main):010:0> def bmi_yp(f,i,p,o)
2 irb(main):011:1>   bmi(feet_to_cm(f,i),
3 irb(main):012:2*       pound_to_kg(p,o))
4 irb(main):013:1> end
5 => nil
```

これでヤード・ポンド法で表わされた身長・体重からも BMI を求めることができる。例えば身長 5 フィート 11 インチ、体重 170 ポンドと書かれているシアトル・マリナーズのイチローであれば次のように計算される。

```
6 irb(main):015:0> bmi_yp(5,11,170,0)
7 => 23.7103429969605
```

(章末の練習問題: 1.9)

1.4.3　局所変数 *

三角形の 3 辺の長さ a, b, c が分かっているとき、その面積は Heron の公式により $\sqrt{s(s-a)(s-b)(s-c)}$ となることが知られている。ただし $s = \frac{1}{2}(a+b+c)$ である。

これを Ruby の関数として定義するときは、変数代入を定義の内側に書いて次のようにできる。

```
1 def heron(a,b,c)
2     s = 0.5*(a+b+c)
3     sqrt(s * (s-a) * (s-b) * (s-c))
4 end
```

ここでは 2 行目で変数 s に $\frac{1}{2}(a+b+c)$ の値を代入し、3 行目はその s を使った式になっている。変数 s は関数 heron の中だけで使われるものなので**局所変数**と呼ばれる。

上の定義の s を展開してしまえば、同じ計算を局所変数を使わずに定義することはできる。しかし局所変数を使えば、式がより簡単になり、局所変数の名前によって式の各部分の意味がより分かりやすくなり、また、同じ計算を何度も行うことが避けられるといった効果がある。

(章末の練習問題: 1.10 1.11)

1.5　ファイルに保存した関数定義

1.5.1　ファイルからの読み込み

キーボードから入力した関数定義は、irb を終了するとすべて消えてしまう。本格的なプログラムを作る場合には、テキストファイルにプログラムを書き、それを irb に読み込ませて実行する。

関数 bmi の例で手順を説明する。まずテキストエディタを使って、次のような内容を入力し、bmi.rb というファイル名で保存する。

この定義の中で、先頭の#で始まっている行は、何を定義しているかの覚え書きであり、プログラムとしては無視される。

```
1 # BMI of a person with height (cm) and weight (kg)
2 def bmi(height,weight)
3   weight / (height/100.0) ** 2
4 end
```

ファイル 1.1: bmi.rb

ファイルを保存するディリレクトリ (フォルダ) は、irb を起動したディレクトリにしておく。ファイル名に付ける.rb という拡張子は、内容が Ruby プログラムであることを示すためのものである。

定義をファイルに保存したら、irb に対してファイルから定義を読み込む命令を与える。

> 注意:　以下では、ファイルから定義を読み込めたことを確認するために、一度 irb を終了して、起動しなおしてから実行している。

```
irb(main):002:0> Control D                    # 一度irbを終了
cm12345$ irb                                  # irbを再起動
irb(main):003:0> load("./bmi.rb")             # ファイル読込
=> true
```

関数 load はファイルから関数定義を読み込むもので Ruby にはじめから用意されている。この関数を、ファイル名を"〜"で囲んだ引数とともに呼び出すと、そのファイル中にある関数定義があたかもキーボードから入力されたかのごとく読み込まれる。ファイル名の先頭には、現在のディレクトリを示す「./」を書く必要がある。

定義が読み込まれていれば、その後は bmi 関数を使うことができる。

```
irb(main):005:0> bmi(188.0, 104.0)
=> 29.4250792213671
```

もし読み込むファイルが irb を起動したディレクトリに保存されていなかったり、ファイル名が違っていた場合は下のような画面になる。この場合は、もう一度ファイル名やファイルのある場所を確認してみよう。

```
irb(main):009:0> load("./bmi.rb")
LoadError: no such file to load -- ./bmi.rb
        from (irb):9:in `load'
        from (irb):9
```

1.5.2　ファイルを読み込むファイル *

第 1.4.2 節で紹介したように、関数定義の中で他の関数を使うことがある。例えばフィートとポンドによる BMI 計算をする bmi_yp は bmi, feet_to_cm, pound_to_kg という 3 つの関数を使っている。このような bmi_yp の定義をファイルに書いてから読み込む場合には、bmi, feet_to_cm, pound_to_kg の定義も読み込んでおかなければならない。

そこで本書では、関数定義をファイルに書く際に、使っている関数を読み込む命令を一緒に書くことにする。つまりある関数 (bmi_yp) を定義するときに、別の関数 (bmi, feet_to_cm, pound_to_kg) を使っていたら、それらの関数が定義されているファイル (bmi.rb, yardpound.rb) を読み込む命令を、ファイルの先頭に書く。

このようにしておけば、bmi_yp を使う場合には、load("./bmi_yp.rb") という命令を実行するだけで必要な定義がすべて読み込まれるようになる。

```
1 load("./bmi.rb")
2 load("./yardpound.rb")
3
4 def bmi_yp(f,i,p,o)
5   bmi(feet_to_cm(f,i), pound_to_kg(p,o))
6 end
```

ファイル 1.2: bmi_yp.rb

1.5.3　誤りの指摘 (エラーメッセージ)

式・命令・定義に誤りがあると、irb は**エラーメッセージ**を表示する。ここまで読み進めた読者はすでに何度か経験済みであろう。誤りの種類は多岐に渡るためすべてを解説することはできないが、典型的な例だけでも見ておこう。

irb に対してキーボードから入力した式や命令が間違っていた場合は、即座にエラーメッセージが表示される。例えば bmi という関数名を誤って bm1 と書いてしまった場合は、次のようなメッセージが表示される。

```
1 irb(main):013:0> bm1(188.0,104.0)
2 NoMethodError: undefined method `bm1' for main:
     Object
3         from (irb):13
```

上の画面の 2 行目は、「メソッド (関数のこと) が見つからない誤り (NoMethodError): bm1 というメソッドは未定義 (undefined)」という意味である。3 行目はその誤りが起きた場所がキーボードからの入力 (irb) の 13 行目に該当する」ということを示している。なお、main:Object というのは変数やメソッドを定義している場所の名前であり、ここでは無視してよい。

またファイル中の定義に誤りがある場合には、それを読み込むときにエラーメッセージが表示されることがある。例えば次のような定義を書いたとしよう。

```
1 def bmi(height,weight)
2   weight / (height/100.0 ** 2
3 end
```

ファイル 1.3: mistake1.rb

この定義をよく見ると、2 行目の (が閉じていない。これを読み込むと次のようなエラーメッセージが表示される。

```
irb(main):003:0> load("./mistake1.rb")
SyntaxError: ./mistake1.rb:3: syntax error,
   unexpected kEND, expecting ')'
        from (irb):3
```

このエラーメッセージの意味は次のようになる。「文法誤り (SyntaxError): mistake1.rb というファイルの 3 行目付近での文法誤りで、) が現われるべき (expecting) ところに、予想外にも (unexpected) end が現われた」ということである。実際の誤りは mistake1.rb の 2 行目にあるのだが、3 行目まで読み込んだ時点でようやく) が抜けていることに気付いたのでこのようなメッセージになっている。このような場合には () の対応関係に問題があるらしいので、その点に注意してプログラムを確認する。

場合によっては、ファイルを読み込む時にはエラーが報告されず、読み込んだ関数を呼び出した時にエラーメッセージが表示されることもある。

```
def bmi(height,weight)
  weight / (hight/100.0) ** 2
end
```

ファイル 1.4: mistake2.rb

このファイル中の定義の誤りがどこだか分かるだろうか。ともかく読み込んでみよう。

```
irb(main):005:0> load("./mistake2.rb")
=> true
```

上のように関数 load を使って読み込んでもエラーメッセージは表示されない。ところが、読み込んだ関数 bmi を使ってみると次のようなメッセージが表示される。

```
irb(main):022:0> bmi(188.0,104.0)
NameError: undefined local variable or method `
   hight' for main:Object
        from ./mistake2.rb:2:in `bmi'
        from (irb):22
```

このメッセージは「名前間違い (NameError): hight という名前の局所変数 (local variable) またはメソッド は未定義 (undefined) である」という意味で、発見場所は上の画面の 5 行目に書かれている情報から./mistake2.rb というファイルの 2 行目の bmi という関数の中である、ということが分かる。

実際、mistake2.rb の定義の 2 行目を見てみると、height を hight と書き間違っていることが分かるだろう。

(章末の練習問題: 1.12)

1.5.4　定数関数

関数 load は関数定義を読み込むことができるが、代入命令によって定義された変数は読み込めない。例えば BMI の計算でいつも h=188.0 と w=104.0 という値を使うので、次のようなファイルを作って読み込んでも変数 h や w の値は読み込まれない。(正確には読み込まれるのだが消えてしまう。)

代入を関数 bmi の定義と一緒に書いておいて、そのファイルを load を使って読み込んでも h や w は定義されない。

```
1  # BMI of a person with height (cm) and weight (kg)
2  def bmi(height,weight)
3    weight / (height/100.0) ** 2
4  end
5  h=188.0          # これらの代入命令はloadされたときに
6  w=104.0          # は効果がない
```

ファイル 1.5: bmi.rb (bmi.rb に代入を追加した後—正しくない例)

代わりの方法として使えるのが**定数関数**を定義することである。例えば BMI の計算であれば、ファイル 1.6 の k_height, k_weight のように引数を 1 つも受け取らずに毎回同じ結果を返すような関数を定義しておく。

```
1  # BMI of a person with height (cm) and weight (kg)
2  def bmi(height,weight)
3    weight / (height/100.0) ** 2
4  end
5  def k_height() #K 選手の身長
6    188.0
7  end
8  def k_weight() #K 選手の体重
9    104.0
10 end
```

ファイル 1.6: bmi.rb (定数関数を追加した後—正しい例)

5 行目や 8 行目の最後にある () は、引数を受け取る変数名を書く所だが、1 つも引数を受け取らないのでこのようになっている。

このファイルを読み込めば、関数 k_height, k_weight を呼び出すことで具体的な数値を書く必要がなくなる。

```
1 irb(main):004:0> load("./bmi.rb")
2 => true
3 irb(main):005:0> k_weight()
4 => 104.0
5 irb(main):006:0> bmi(k_height(), k_weight())
6 => 29.4250792213671
```

これらの関数は引数を 1 つも受け取らないので、呼び出す際には k_height() のように、「空の引数」を書くことになる。

もちろん 188.0 と書いた方が k_height() と書くよりも短いので、このやり方ではキーボードの入力回数を減らしたことにはならない。しかし、名前の付いた定数関数を使うことによって具体的な値を覚えておく必要がなくなり、数値の書き間違いを防ぐこともできる。特に円周率やプランク定数のような桁数の多い数値を使う場合には、キーボードから何桁も入力することは大変なので、定数関数が活躍する。

1.6　定義のまとめ

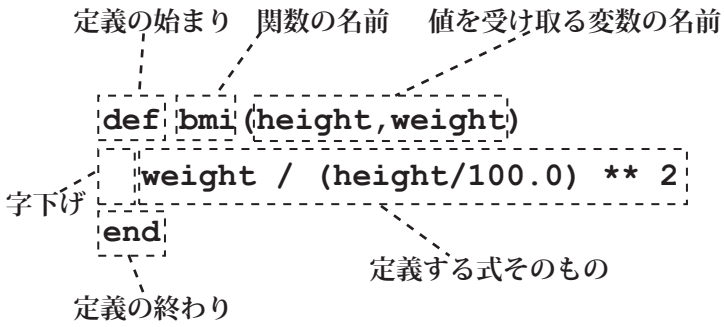
この章で紹介した式や命令をまとておこう。

include(Math)： cos や sqrt などの数学関数を使う前に、実行しておかなければいけない命令。

変数名 = **式**： 右辺の式を計算した値を、左辺に書かれた名前の変数にしまう**代入命令**。

関数名(式$_1$,..., 式$_n$)： 式$_1$,..., 式$_n$ を計算した値を、関数に渡す**関数呼び出し式**。関数に渡される式の値のことを**引数**(ひきすう)という。関数名 は sqrt のような予め定義されている関数でも、bmi のような自分で定義した関数でもよい。

def **関数名**(変数名$_1$,..., 変数名$_n$) ↵ 式 ↵ end： **関数定義**。少し複雑なので、具体例で説明する。



`def` と `end` は、定義の範囲を示している。**関数の名前**(ここでは `bmi`) は、変数名と同様、好きな名前を付けることができる。次の `(height,weight)` は、関数が値を受け取るために使う変数名である。

関数を「定義する式そのもの」では `height`, `weight` という変数を使うことができる。またこの式の先頭には、数文字の空白を入れて `def` と `end` の対応を見つけやすくしている。これは**字下げ**といって、必ずしも必要ではないのだが、見やすさのために読者も行うようにして欲しい。

`load("`**ファイル名**`")`: ファイルから関数定義を読み込む命令。`irb` を起動したディレクトリに置かれているファイル名を読み込む場合には `"./bmi.rb"` のように先頭に「`./`」を付ける。

1.7　章末問題

練習 1.3 (もっと数式の計算) 次の計算を行う Ruby の式と、その結果を書け。

a) 平成 16 年の 19 歳の日本人の平均身長・体重は以下の通りであった。それぞれの BMI を求めよ。

「平成 16 年国民健康・栄養調査報告」より。

	身長 (cm)	体重 (kg)
男	171.2	59.9
女	158.7	51.9

b) セ氏 22 度を華氏温度に変換した値。ただしセ氏温度 C と華氏温度 F は $F = \frac{9}{5}C + 32$ という関係

c) 華氏 50 度をセ氏温度に変換した値

d) 東京大学の授業料 535800 円の米ドル換算値。ただし 1 米ドルは 116.16 円とせよ

2008 年 8 月現在もの。`http://www.u-tokyo.ac.jp/stu01/e03_j.html` より。

e) Stanford 大学の授業料 12010 ドルの円換算値。

2008-9 年度のもの。`http://registrar.stanford.edu/students/finances/` より。

f) 中国におけるビッグマックの値段 11 元の日本円換算値。ただし 1 米ドルは 7.512 元とせよ。

g) 2005 年に米国に上陸したハリケーン Katrina の最大風速毎時 175 マイルの毎秒メートル換算値。1 マイルは 1609 メートルである。

h) 10 年後の日本の国債残高の 1 人あたりの額。ただし、日本が発行している普通国債の平均利率は 1.41%, 残高は 5414584 億円だとせよ。また、利息は複利法で加わり、追加発行も返済もしないとする。日本人の人口は 1.26 億人とする。

財務省「普通国債の利率別現在額と残高合計に占める割合（平成 20 年 3 月末）及び利率加重平均」より。

練習 1.4 (数学関数) 以下の計算を行う式と結果を書け。(注: 複数の関数を組み合わせなければいけない場合もある。)

a) $\sqrt{10},\ (\sqrt{2}\sqrt{5}), \sqrt{\sqrt{5}}\ \sqrt[12]{2}$

b) $\sin 30°,\ \cos 30°,\ \tan 30°$

c) $\log 1000,\ \log_{100} 100,\ \log_2 1000$

d) 2.7^{10} (** を使わずに)

e) テクネチウム 99m は、原子 1 個あたり 1 秒間に 0.0000320362394891 個の割合で崩壊する。半減期を求めよ。

練習 1.5 (変数の利用) 練習 1.3 で考えたそれぞれ式について、後から変化させることがありそうな値がどれかを考えよ。そのような値のかわりに変数を使って式を書き直せ。

練習 1.6 (代入) 次の計算を代入を使って行え。

a) $x = 10$, $y = x(x-3)$, $z = y(y-3)$ としたときの $z(z-3)$ の値。またもし変数を使わずに式を展開したらどのような式になるか。

b) 以下は 2 次方程式に関する一連の計算である。

   (a) $a = 3,\ b = 5, c = -7$ とする。

   (b) 二次方程式 $ax^2 + bx + c = 0$ の判別式の値を d とする。

   (c) p, q を $ax^2 + bx + c = 0$ の 2 つの解とする。

   (d) このときの $ap^2 + bp + c$ および $aq^2 + bq + c$ の値。

練習 1.7 (代入の右辺の式) 代入の右辺は `r=1/116.16` のように数式であってもよいし、その数式に変数が使われていてもよい。右辺が数式だった場合は、その式を計算した値が変数に代入される。

さて、米国 the Centers for Disease Control and Prevention の報告では 2002 年の 20 歳から 74 歳の米国人の平均身長、体重は以下の通りであった。

男女それぞれの BMI を計算せよ。このとき BMI の計算式自体は上と同じものになるように変数 h, w には cm, kg に換算した値を代入するように命令を書け。

	身長	体重
男	5 フィート 9.5 インチ	191 ポンド
女	5 フィート 4 インチ	164.3 ポンド

練習 1.8 (もっと関数の定義) 次のような関数を定義せよ。さらに、それを使った計算の例を示せ。

a) セ氏温度 c を華氏温度に変換する `celsius_to_fahrenheit(c)`.

b) 華氏温度 f をセ氏温度に変換する `fahrenheit_to_celsius(f)`.

c) 毎秒メートル (m/s) で表わされた速度 v を毎時マイルに変換する `ms_to_mph(v)`.

d) 上と逆の変換をする `mph_to_ms(v)`.

e) 米国では気温 (華氏 t 度) 風速 (v 毎時マイル) を加味した体感温度 (Wind Chill Index) を次のような式で定めている。

$$35.74 + 0.6215\mathtt{t} - 35.75(\mathtt{v}^{0.16}) + 0.4275\mathtt{t}(\mathtt{v}^{0.16})$$

この変換をする関数 `wind_chill_index(t,v)`. (例えば 2007 年の 10 月 20 日のニューヨークは華氏 20 度、風速 20 毎時マイルであった。)

f) 上で定義されている体感温度を、気温がセ氏、風速が m/s で与えて、結果もセ氏で求めたい。そのような関数 `wind_chill_index_celsius(t,v)`.

g) 二次方程式 $\mathtt{a}x^2 + \mathtt{b}x + \mathtt{c} = 0$ に関して

(a) 判別式 $\mathtt{b}^2 - 4\mathtt{ac}$ を求める `det(a,b,c)`.

(b) 解の 1 つ $\frac{-\mathtt{b}+\sqrt{\mathtt{b}^2-4\mathtt{ac}}}{2\mathtt{a}}$ を求める `solution1(a,b,c)`. (`det` を使って定義せよ。)

(c) もう 1 つの解 $\frac{-\mathtt{b}-\sqrt{\mathtt{b}^2-4\mathtt{ac}}}{2\mathtt{a}}$ を求める `solution2(a,b,c)`. (`solution1` と `solution2` の共通部分を 1 つの補助関数にできるか?)

(d) 二次関数 $f(\mathtt{x}) = \mathtt{a}\mathtt{x}^2 + \mathtt{b}\mathtt{x} + \mathtt{c}$ の値を求める `quadratic(a,b,c,x)`.

練習 1.9 (ヤード・ポンド法 BMI の完成) 練習 1.2b と練習 1.2c の定義を `yardpound.rb` というファイルに書け。新たに起動した `irb` で `load("./bmi_yp.rb")` を実行した後で `bmi_yp` を使った計算ができることを確めよ。

練習 1.10 (局所変数の有効範囲 *) 第 1.4.3 節で定義した関数 `heron` の外で変数 s が使われていた場合、関数呼び出しの後の s の値はどうなっているだろうか。次の命令と式を順に実行・計算し、関数定義の中と外で使われている変数 s の関係について説明を試みよ。

```
s = 123
heron(20,30,40)
s
```

練習 1.11 (局所変数と補助関数 *) ヒラメキの良い市谷くんは局所変数を定義せずに heron と同じ計算をする関数が定義できたと主張している。

青山さん:「疑わしいわね。s の定義を展開したんじゃないの?」

市谷くん:「そんなことはないさ。0.5*(a+b+c) という計算は一度しかやらないよ。補助的な関数を定義するのが鍵なんだ。」

市谷くんが定義した heron はどのようなものか、作ってみよ。

練習 1.12 (わざと間違ってみる) 第 1.5.3 節で紹介した他にも、誤りの種類は数多くある。実際にエラーメッセージを見てから誤りを探すことは時として簡単でない。練習のために、意図的に誤りを混入させたファイルを作り、どのようなエラーメッセージが表示されるかを観察しておけ。後日、同じようなエラーメッセージが表示されたときの原因究明に役立つはずである。例えば、次のような誤りを含んだファイルを作り、読み込ませてみよ。

a) ファイル 1.1 の 1 行目の # 記号を取り除いたもの。

b) 空白の挿入。定義の中の適当な位置に空白文字を入れてみよ。場所によっては誤りになるし、場所よってはならない。

c) 改行の挿入。定義の中の適当な位置で改行してみよ。場所によっては誤りになるし、場所よってはならない。

d) end の不足。定義の終わりを示す end を取り除いてみよ。できれば、複数の関数定義があるファイルで、最初の定義の end を取り除いてみよ。

e) いわゆる全角文字、つまり、日本語文字コードが混ざったもの。例えば空白を全角空白に置きかえたものや、数字を全角の数字に置きかえたものを試みよ。

第2章　配列による画像の表示

ここまでの計算は数値を扱うものだけであった。現在のコンピュータは画像や音声のようなデータ、ワードプロセッサや電子メールのように文字列データなどを扱うこともできる。ここでは画像データの表現方法と、それを一般化した**配列**について見てゆこう。

2.1　画像の表現

コンピュータにおける画像の表現方法は、画像を小さなマス目に区切り、各マス目が塗りつぶされているかどうかという情報を並べて表わすのが一般的である。例えば「  」という画像を拡大してみると図 2.1 のように 8×8 のマスの何箇所かが塗りつぶされているかどうかによって表現されていることが分かる。

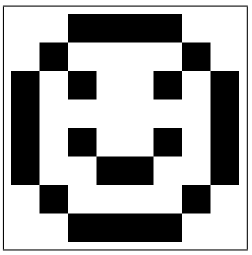


図 2.1: 画像の例

プログラムの中からこのような画像を表示する方法は一通りではなく、プログラミング言語によっても異なる。本書では画像を簡単に表示するための機能を組み込んだ `isrb` というソフトウェアを `irb` のかわりに使用するので、それに従って説明する。

まず `isrb` を使うために、一度 `irb` を終了し、`isrb` という名前のソフトウェアを起動する。

```
1 irb(main):002:0> Control D
2 cm12345$ isrb ↵
3 >> ▮
```

この `isrb` は本教科書のために作成されたソフトウェアであり、Ruby 処理系に付属しているものではない。一般のコンピュータで利用するための情報については本書冒頭 (p.1) に記載されている web ページを参照されたい。

ソフトウェアの都合上、プロンプトが「irb(main):001:0>」のようなものから「>>」に変わったが、irb と全く同じように式や命令を入力して計算・実行させることができる。

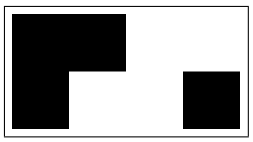


図 2.2: 簡単な画像

早速、図 2.2 のような画像を表示させよう。isrb では、この画像をマス目の白い部分を 1, 黒い部分を 0 として次のような式で表わす。

```
1 >> a = [[0,0,1,1],
2 ?>       [0,1,1,0]]
3 => [[0, 0, 1, 1], [0, 1, 1, 0]]
```

上の画面で変数 a に代入されている右辺の式が 2×4 の画像データである。式が 2 行に渡っていることに注意せよ。なお、白が 1 で黒が 0 であるのは、マス目の明るさ (明度) を 0 から 1 までの数値で表わしているためである。

このデータを表示するには、isrb の関数 show を使う。

```
4 >> show(a)
5 => nil
```

これで図 2.2 のような画像が表示される。

練習 2.1 (画像データの作成)　a) 図 2.1 のような画像データを作成し、表示する命令を作れ。

b) 次のようなデータを作成し、画像として表示させよ。

```
1 w = [[0,1,1,1,1,1],
2      [0,1,0,0,0,1],
3      [0,1,0,1,0,1],
4      [0,1,1,1,0,1],
5      [0,0,0,0,0,1]]
```

2.2　画像の操作

画像データは表示するだけでなく、大きさに関する情報を得たり、各点 (マス目) に関する情報を得たり変更したりするといった基本的な操作を行うことができる。各点の明度は、画像の左上の座標を (0,0) として次のような式で調べることができる。

```
>> a[0][0]        #座標(0,0)の明度を参照
=> 0
>> a[0][2]        #座標(2,0)の明度を参照
=> 1
```

式としては Y 座標、X 座標の順に書くことに注意せよ。この順序は奇異に思えるかも知れないが、数学で行列中の要素を「2 行 3 列」のように行番号を先に書くのと同じだと思っておけばよい。

画像データの各点の明度を変更することもできる。

```
>> a[1][2]=0.5   #座標(2,1)の明度を変更
=> 0.5
>> show(a)       #再表示
=> nil
```

10 行目は、「画像 a の座標 (2,1) の点」に 0.5 を代入するという意味になる。この命令を実行した後は、図 2.3 のように座標 (2,1) の点の明度が半分、つまり灰色になる。

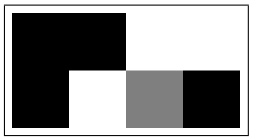


図 2.3: 変更後の画像

画像データの内容を変更しただけでは画面に表示されている画像は変化しないが、12 行目のようにもう一度 show を実行すると変更された内容に表示が変わる。

練習 2.2 (画像データの操作) 画像データの座標 (x, y) の点と、その周囲の 8 点の合計 9 点の明度の平均値を計算する関数 image_average9(image,x,y) を作れ。定義を簡単にするために image は画像データ、座標 (x, y) は常に画像の内側の点になっていると仮定してよい。つまり image の幅と高さが h, w だったとき、$1 \leq x \leq w-2$,　$1 \leq y \leq h-2$ だと仮定してよい。(整数を整数で割ると値が切り捨てられてしまうので、そうならないように工夫せよ。)

```
irb(main):012:0> b = [[0,1,0],
irb(main):013:1*       [1,0,1],
irb(main):014:1*       [0,1,0]]
=> [[0, 1, 0], [1, 0, 1], [0, 1, 0]]
irb(main):015:0> image_average9(b,1,1)
=> 0.444444444444444
```

作成した関数を使って練習 2.1a で作成した画像の座標 (4,3)、座標 (5,2) を中心とした明度の平均値を計算せよ。

(章末の練習問題: 2.3)

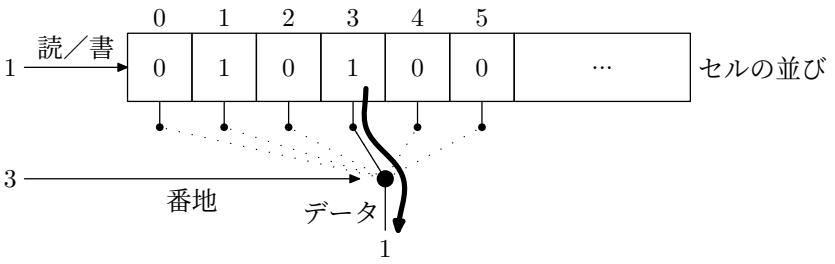
2.3　カラー画像の表現 *

isrb に用意されている関数 show は、カラー画像も表示できる。次の例は 3 × 2 のカラー画像を表示する。

```
1 >> d=[[[0,0,0],[0,1,0],[0,0,1]],
2 ?>     [[1,0,0],[1,1,0],[1,0,1]]]
3 => [[[0, 0, 0], [0, 1, 0], [0, 0, 1]], [[1, 0, 0],
     [1, 1, 0], [1, 0, 1]]]
4 >> show(d)
5 => nil
```

括弧が多くて分かりづらいが、配列 d の大きさは 2 × 3 × 3 の 3 次元配列である。いままで 1 つの点は 0 から 1 までの明度によって表わされていたが、カラー画像の場合は、赤、緑、青の 3 原色の明度を表わす大きさ 3 の配列にな

ここで紹介した画像を表わすデータは、一般に**配列**と呼ばれ、コンピュータの主記憶装置を抽象化したものである。今日のコンピュータの主記憶装置は、ランダムアクセスメモリ (RAM) という電子回路によって構成されている (「情報」(川合編, 東京大学出版会)7 章参照。)。RAM は、1 ビットの情報を記憶する**セル**を沢山並べ、1 つのセルを**番地**と呼ばれる整数によって指定して読み書きできるようにした回路である (下図)。テープやハードディスクのように頭出しが必要な記憶装置と異なり、どの番地のセルに対しても即座に読み書きができる点が特徴である。配列は添字を指定して値を読み書きできるデータであるが、基本的には添字の大小に関らずに即座に読み書きができる点で RAM と同じ特徴を持っている。



る。このような表現方法を、red, green, blue の頭文字から **RGB 画像**と呼ぶ。このとき、画像の座標 (x, y) の赤成分は `d[`y`][`x`][0]`, 緑成分は `d[`y`][`x`][1]`, 青成分は `d[`y`][`x`][2]` になる。例えば座標 $(2, 1)$ の色は `d[1][2][0]` は 1, `d[1][2][1]` は 0, `d[1][2][1]` は 1 なので、赤と青を合成した紫になる。

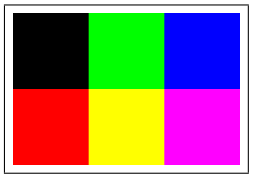


図 2.4: カラー画像

(章末の練習問題: 2.4)

2.4　定義のまとめ

配列を作る: `[` 式$_0$`,` 式$_1$`,` ..., 式$_{n-1}$ `]` という式によって大きさ n の配列を作ることができる。作られた配列の i 番目には 式$_i$ の値が入っている。例えば `[ 1+2, 3+4 ]` という式は大きさ 2 の配列を作る。

配列の参照: 式$_0$`[` 式$_1$`]` という式は、式$_0$ が表わす配列の 式$_1$ 番目の値を参照する。このような配列中の要素の位置のことを**添字**(そえじ)という。例えば `a[1]` は変数 `a` に代入されている配列の 1 番目を参照する。

添字が配列の範囲外だった場合には、値がないことを示す `nil` という特別な値を得る。

配列の中身の変更: 式$_0$`[` 式$_1$`]=` 式$_2$ という命令は、配列の先頭を 0 番目として、式$_0$ が表わす配列の 式$_1$ 番目の値を 式$_2$ の値に変更する。例えば `a[1] = 5*6` は、変数 `a` に代入されている配列の 1 番目を 30 に変更する。

配列の大きさ: 式`.length()` という式は、式 が表わす配列の大きさを求める。

高次元の配列: 配列を作る式の中に配列を作る式を書くと、2 次元以上の配列を作ることができる。例えば `m = [ [ 0, 1, 2 ], [ 3, 4, 5 ] ]` という命令は、2 行 3 列の配列を作り、`m` に代入する。参照は `m[1][2]` のように、参照を行う `[ ]` を並べて書く。

配列の表示: `show(` 式 `)` という命令は、式 が表わす配列 `a` の内容を画像として表示する。配列の中身は 0 から 1 までの数値でなければいけない。

配列が 2 次元のときは、a[y][x] の数値が座標 (x, y) の点の明るさに対応した濃淡画像が表示される。3 次元のときは a[y][x][c] の数値が座標 (x, y) の点の c 番目の原色の明るさに対応したカラー画像になる。ただし c は 0 が赤、1 が緑、2 が青に対応している。

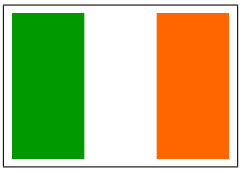
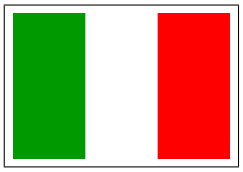
なお、show は isrb の中からでないと使用できない。

2.5　章末問題

練習 2.3 (配列)　a) a は数値が並んだ 1 次元の配列とする。a の x 番目の値と、その前後の値の合計値を求める sum3(a,x) を作れ。ただし x の値は 1 以上かつ (a の大きさ − 1) 未満であり、必ず前後に値があるものと仮定せよ。例えば a が [1,2,3,4] のとき sum3(a,1) は 6 に、sum3(a,2) は 9 になる。

b) a の x 番目の値と、その前後の値の平均値を求める array_average3(a,x) を作れ。ただし x の値は 1 以上かつ (a の大きさ − 1) 未満であると仮定せよ。例えば a が [1,2,3,4] のとき array_average3(a,1) は 2.0, array_average3(a,2) は 3.0 になる。

練習 2.4 (国旗) カラー画像表現を使って簡単な国旗を描いてみよ。例:


第3章　条件分岐と繰り返し

この章では数値以外のデータとして**真偽値**と**文字列**を紹介する。実用的なプログラムは、複雑な判断を行いながら処理を行うことが多いが、この判断部分の計算に使われるのが**真偽値**である。また、利用者とのデータのやりとりのために文字情報を扱うことは多いが、そこで用いられるのが**文字列**である。

この章ではまた、沢山のデータを処理する方法の手始めとして、配列に対する**繰り返し**を紹介する。繰り返しによって規則的な画像を生成することや、統計処理など様々な処理を行うことができる。

3.1　条件分岐

3.1.1　場合分けを使った計算

場合分け、つまり「ある場合にはこの値、別の場合にはこの値」といった計算を行いたいことは少なくない。例えば「2人の得点 x, y の最高点 $\max(x, y)$」を求める問題は「$y < x$ の場合は x, そうでなければ y」という計算だと言える。他にも「x の絶対値 $\mathrm{abs}(x)$」($x > 0$ かどうか) や「2次方程式 $ax^2 + bx + c = 0$ の実数解の個数 $\mathrm{solutions}(a, b, c)$」(判別式の正負) のような計算もやはり場合分けである。

このような場合分けは、数学では次のように書くことができる。

$$\max(x, y) = \begin{cases} x & (y < x \text{ のとき}) \\ y & (\text{それ以外}) \end{cases}$$

プログラミング言語では**条件分岐**を使って、場合分けのある計算を表わす。例えば max であれば次のように定義できる。

上の定義の 2 行目から 6 行目までが場合分けをしている 1 つの式である。`if`, `else`, `end` が区切りとして使われているが、数学的な書き方との対応は自明だろう。

以下は実際に使ってみた様子である。

```
1 irb(main):003:0> load("./max.rb")
2 => true
3 irb(main):004:0> max(123, 456)
4 => 456
```

```
1 def max(x,y)
2   if y < x
3     x
4   else
5     y
6   end
7 end
```

ファイル 3.1: max.rb

```
5 irb(main):005:0> max(max(12, 34), max(56, 78))
6 => 78
```

練習 3.1 (絶対値) 数値 x の絶対値を求める関数 abs(x) を定義せよ。

3.1.2　3 通りの場合分け

ある値 x の符号、つまり x の正負に応じて $1, -1, 0$ のどれかを答えるような関数 $\mathrm{sign}(x)$ を考えよう。数学であれば次のような 3 つの場合分けによって書ける。

$$\mathrm{sign}(x) = \begin{cases} -1 & (x < 0) \\ 1 & (0 < x) \\ 0 & (\text{それ以外}) \end{cases}$$

条件分岐は 1 つの条件を調べ、それが成り立つ場合とそうでない場合に分けるのが基本なので、3 つの場合に分けるには**入れ子**にする必要がある。つまり、条件分岐によって場合分けされた式の中に、さらに条件分岐を書く必要がある。

(章末の練習問題: 3.5)

3.1.3　複雑な条件

現実世界の問題は、かなりこみ入った場合分けを行わなければいけないこともある。原理的には if y < x という形の条件分岐だけを何重にも入れ子にすることで、かなり複雑な場合分けを行うことも可能ではある。しかし、そうやって出来あがった条件分岐がどんな場合分けをしているのかを理解することは簡単でなくなってしまう。

そのため Ruby をはじめとする多くのプログラミング言語には、色々な種類の比較を行うための記号と、条件式を組み合わせるための記号が用意され

```
1  def sign(x)
2      if x < 0
3          -1
4      else
5          if 0 < x
6              1     # not(x<0) and 0<x
7          else
8              0     # not(x<0) and not(0<x)
9          end
10     end
11 end
```

ファイル 3.2: sign.rb

ており、複雑な条件を 1 つの条件式として書けるようになっている。より詳しくは第 3.2 節で説明するが、次のようなものが代表的である。

色々な比較

数の大小および、等しさの比較を行う等号と不等号には次のようなものがある。いずれも数学における等号・不等号に似せている。

書き方	数学	意味
`x > y`	$>$	x が y より大きい
`x >= y`	$\geq$	x が y 以上
`x == y`	$=$	x と y が等しい（x=y でないことに注意）
`x < y`	$<$	x が y より小さい
`x <= y`	$\leq$	x が y 以下
`x != y`	$\neq$	x と y が異なる

条件式の組み合わせ

条件式を組み合わせてより複雑な条件式を作ることができる。Ruby では ||, &&, ! という記号を使って「A または B」「A かつ B」「A でない」という条件を表わすことができる。以下に例を示す。

書き方	意味
`x > y || x == 0`	`x > y` または `x == 0`
`x < y && y < z`	`x < y` かつ `y < z`
`!(x < y && y < z)`	`(x < y` かつ `y < z)` でない

(章末の練習問題: 3.6)

3.2　真偽値を与える論理演算

第 3.1 節では `if y < x` のように、数の大小などに応じて場合分けをする方法や、複数の条件を `&&` などで組み合わせることで、より複雑な場合分けができることを見た。ここでは、そこで使われた条件式を、**真偽値**に関する**論理演算**として整理しよう。

まず**真偽値**とは、**真**つまり「正しい」と**偽**つまり「正しくない」という 2 つの値のことである。Ruby では真を表わす値を `true`, 偽を表わす値を `false` と書く。

条件式 `1 < x` は、2 つの数値を比較して真偽値を求める式である。数式 `1 + x` が 2 つの数値の合計値を求める式であることと対比させると、結果の値の種類が違うことを除けばよく似ている。実際、`1 < x` のような式を単独で計算することもできる。

```
irb(main):003:0> x = 3
=> 3
irb(main):004:0> 1 < x
=> true
irb(main):005:0> x == 2
=> false
```

また、答えが真偽値となるような関数を定義することもできる。例えば「ある数 x が偶数である」ことは、x を 2 で割った余りが 0 かどうかで判定できるので、次のような関数として定義することができる。

```
def is_even(x)
  x%2 == 0
end
```

ファイル 3.3: `is_even.rb`

このように定義した関数は、条件式の中で使うことができる。例えば、次のような関数 $\mathrm{tnpo}(n)$ を考える。

この関数名は、奇数の場合の式の英語読み “three n plus one” からとっている。

$$\mathrm{tnpo}(n) = \begin{cases} \frac{n}{2} & (n \text{ が偶数}) \\ 3n+1 & (n \text{ が奇数}) \end{cases} \tag{3.1}$$

これを Ruby の関数として定義する際には、「`n` が偶数」かどうかを判定する必要が生じるが、上で定義した `is_even(x)` を使えば次のように定義できる。

論理演算子`&&`, `||`, `!` (第 3.1.3 節) は、1 つまたは 2 つの真偽値から真偽値を求める演算である。これらを使った式も単独で計算できる。

```
irb(main):007:0> true && 1 < x
```

```
load("./is_even.rb")

def tnpo(n)
  if is_even(n)
    n/2
  else
    3*n + 1
  end
end
```

ファイル 3.4: tnpo.rb

```
=> true
irb(main):008:0> false || true && !(false||true)
=> false
```

条件を組み合わせた論理式は、一見して何を判定しているのかが分かりづらいので、関数として定義し、意味を明確にすることは重要である。例えばある数 x が 0 に近いことを判定する near_zero(x) は次のように定義できる。
(章末の練習問題: 3.7 3.8)

3.3　文字列

名前や文章のような情報は、プログラム中では文字を 1 列に並べた**文字列**として扱われる。文字列は、各文字の**文字コード** に対応する整数が沢山並んでいるものなので、配列の一種とも言える。実際、「先頭から n 文字目を参照する」とか「長さを調べる」といった配列と同様の操作が用意されている。

「情報」(川合編, 東京大学出版会)p.19 参照。

本書では簡単のために英語のアルファベット、数字、記号のいわゆる ASCII 文字集合だけからなる文字列のみを扱う。以下、具体例を用いて文字列の扱いを簡単に説明する。

例として「abra」と「cadabra」という言葉を操作してみよう。このような言葉を値として扱うためには二重引用符 (") 記号で囲めばよい。

実はこの書き方は load 命令に与えるファイル名のところですでに見ている。

関数 tnpo(n) は練習 4.12c, 練習 4.8c で紹介する Collatz 予想に登場する計算である。tnpo(7), tnpo(tnpo(7)), tnpo(tnpo(tnpo(7))), ... を計算してみよ。

```
1  def epsilon()
2    0.0000001
3  end
4
5  def near_zero(x)
6    -epsilon() < x && x < epsilon()
7  end
```

ファイル 3.5: near_zero.rb (この定義では充分に小さな値 ε として定数関数 (第 1.5.4 節) epsilon() を定義して使っている。)

```
1  irb(main):003:0> s = "abra"
2  => "abra"
3  irb(main):004:0> t = "cadabra"
4  => "cadabra"
```

これで変数 s と t に「abra」と「cadabra」という文字列がしまわれた。2 つの**文字列をつなげる**には、+ 記号を使う。

```
5  irb(main):006:0> u = s + t
6  => "abracadabra"
7  irb(main):007:0> "123" + "456"
8  => "123456"
```

数値の足し算と同じ記号を使っているが、文字列の場合はつなげた文字列を作る。従って 7 行目のように 123 という**文字列**に 456 という**文字列**を+した結果は 123456 という文字列になる。

文字列処理では、色々な長さの文字列を扱うことが多い。例えば、名簿にある名前の長さは人それぞれである。そのような場合には、それぞれの文字列の長さを調べ、その長さに応じた処理をすることになる。

文字列の長さを調べる文字列を表わす式の後に.length() と書く。

```
9  irb(main):009:0> s.length()
10 => 4
11 irb(main):010:0> (s + t).length()
12 => 11
```

文字列の一部分を取り出すには次のようにする。

```
13 irb(main):012:0> s[0..0]
14 => "a"
```

```
15 irb(main):013:0> s[1..2]
16 => "br"
```

ここで「..」の前後は、取り出す最初の文字の添字と最後の文字の添字を意味する。

(章末の練習問題: 3.9)

3.4　繰り返しによる画像の作成

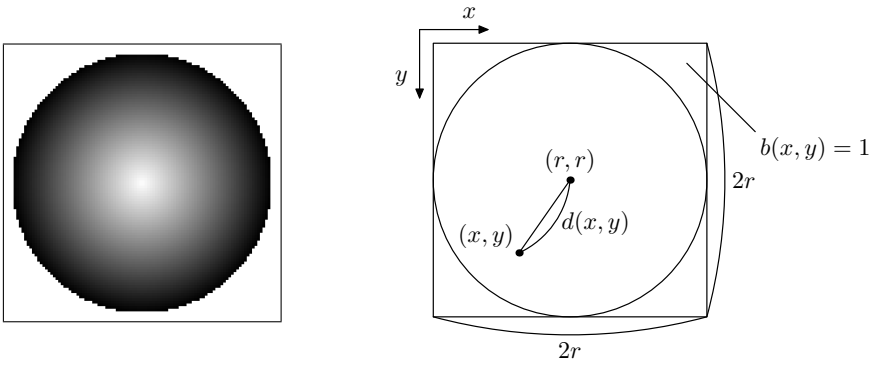


図 3.1: 繰り返しによって作成した画像 (左) とその位置関係 (右)

図 3.1(左) のような画像を作ることを考えよう。一見難しそうに思えるかもしれないが、各点について画像の中心からの距離を求め、それに従って明度を決めているだけなので、かなり規則的なものである。具体的には、図右のように画像の大きさを $2r \times 2r$, 座標 (x, y) の点 (r, r) からの距離を $d(x, y)$ としたときに、点 (x, y) の明度 $b(x, y)$ を以下のように決めている。

$$b(x,y) = \begin{cases} \frac{r-d(x,y)}{r} & (d(x,y) \le r) \\ 1 & (d(x,y) > r) \end{cases} \tag{3.2}$$

問題は、どうやって大きさ $2r \times 2r$ の配列を作り、中身を上の計算式で定めてやるかである。このためには**繰り返し**命令が必要になる。順にやり方を見てゆこう。

3.4.1　与えられた大きさの 1 次元配列を作る

これまで配列を作る場合は、 `[1,2,3]` のような式を書いていた。この場合、`[ ]` の内側の式の個数がそのまま配列の大きさになった。この方法では、予め決められた大きさの配列しか作れない。

与えられた大きさの配列を作るには Ruby では `Array.new(10)` のような式を書く。10 の部分は任意の式で置き換えることができる。

> 注意:　以降では作成した配列を画像として表示するので、`irb` ではなく第 2.1 節で説明した `isrb` を使う。もし `irb` を使っている場合は一度終了して `isrb` を起動しなおしておくこと。

```
1 >> image = Array.new(6)
2 => [nil, nil, nil, nil, nil, nil]
3 >> a = Array.new(3)
4 => [nil, nil, nil]
```

作られた配列は `[1,2,3]` のような式によって作られたものと同様に扱える。ただし、作られた当初の中身は `nil` という「空である」ことを示す特別な値になっている。

3.4.2　繰り返しによって、配列の全要素をそれぞれ変更する

次に配列の要素を規則的に決める方法である。数学風には

$$a_i = 0 \qquad (\forall i \in 0 \ldots 2)$$

と書いて「数列 a の要素 a_0 から a_2 までがすべて 0 である」ことを表わすが、Ruby ではこれに似た `for` 命令を用いて、配列 `a` の全要素を 0 に変更できる。

```
5 >> for i in 0..2
6 >>   a[i] = 0
7 >> end
8 => 0..2
9 >> a
10 => [0, 0, 0]
```

上の画面の 5 行目から 7 行目は、「`i` を 0,1,...,2 の順に変化させて、毎回 `a[i]=0` を実行せよ」という**繰り返し命令** `for` である。この命令を実行した後の `a` を見ると (9 行目) 確かにすべてが 0 になっている。

練習 3.2 (0 ばかりの配列を作る) 大きさ `n` で中身がすべて 0 であるような 1 次元配列を作る関数 `make1d(n)` を定義せよ。(ヒント: 3 行目から 7 行目までがほぼそのまま関数定義の中身になる。)

```
1 >> make1d(10)
2 => [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

(章末の練習問題: 3.11)

3.4.3　2 次元配列を作る

2 次元の配列は、すべての中身が 1 次元の配列であるような 1 次元の配列なので、6 行 3 列の配列を作りたかったら練習 3.2 で作った make1d を使って次のようにすればよい。

```
>> for i in 0..5
>>     image[i] = make1d(3)
>> end
=> 0..5
>> image
=> [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0,
    0, 0], [0, 0, 0]]
```

練習 3.3 (2 次元配列) h 行 w 列の配列を作る make2d(h,w) を定義せよ。ただし、作られる配列の中身はすべて 0 にせよ。

```
>> make2d(2,3)
=> [[0, 0, 0], [0, 0, 0]]
```

3.4.4　2 重の繰り返し

2 次元の配列を作ったら、あとはすての x, y の組み合わせについて image[y][x] = b(r,x,y) のような代入をすれば図 3.1 のような画像を作ることができる。

このときすべての x, y の組み合わせについて代入をするためには、**2 重の繰り返し**を行う。次の関数定義 sphere を見てほしい。

```
def sphere(r)
  image = make2d(2*r, 2*r)
  for y in 0..(2*r-1)
    for x in 0..(2*r-1)
      image[y][x] = b(r,x,y)
    end
  end
  image
end
```

ファイル 3.6: sphere.rb

この定義では 3 行目から 7 行目までが y を変化させる繰り返しである。その内側の 4 行目から 6 行目 までは、x を変化させる繰り返しなので、

- y を 0 として、x を $0, 1, 2, \ldots, 2\mathtt{r}-1$ と変化させて 5 行目を実行
- y を 1 として、x を $0, 1, 2, \ldots, 2\mathtt{r}-1$ と変化させて 5 行目を実行
- y を 2 として、x を $0, 1, 2, \ldots, 2\mathtt{r}-1$ と変化させて 5 行目を実行

$\vdots$

- y を $2\mathtt{r}-1$ として、x を $0, 1, 2, \ldots, 2\mathtt{r}-1$ と変化させて 5 行目を実行

という順序で実行が起き、結果として画像のすべての点の明度を決めることができる。

ここでの 5 行目の右辺は p. 37 の式 3.2 (に対応する関数 b) であるが、この式を適当なものに変えることで色々な図形を作成することができる。

練習 3.4 (繰り返しによる画像の作成) 式 3.2 の計算をする関数 `b(r,x,y)` を定義せよ (x, y だけでなく r も引数となっていることに注意せよ)。`show(sphere(20))` を実行して図 3.1 のような画像が表示されることを確めよ。

(章末の練習問題: 3.12 3.13)

3.5　定義のまとめ

条件分岐: 式$_1$ が成り立つときは 式$_2$ を、そうでないときは 式$_3$ を計算する条件分岐は次のように書く。このとき 式$_1$ を**条件式**という。

```
if 式1
    式2
else
    式3
end
```

真偽値: `true` と `false` はそれぞれ、真と偽を表わす値である。

文字列を作る: 2 つの " 記号の間にある文字や記号は文字列を作る。

文字列の結合: 式$_1$ + 式$_2$ という式は、式$_1$, 式$_2$ の値が文字列のとき、それらの文字列をつなげた文字列を作る。

文字列の長さ: 式$_1$`.length()` という式は、式$_1$ が表わす文字列の長さを求める。

部分文字列: 式$_1$`[`式$_2$`..`式$_3$`]` という式は、式$_1$ が表わす文字列の 式$_2$ 番目から 式$_3$ 番目までを取り出した文字列を作る。

空の値: `nil` は「無い」ことを表わす特別な値である。

配列の作成: `Array.new(`式`)` という式は、大きさが式の値であるような配列を作る。作られた配列の中身はすべて `nil` である。

繰り返し: 変数の値を式$_1$の値から式$_2$の値まで 1 ずつ順に変化させながら、命令$_1$から命令$_n$を毎回実行する**繰り返し**は次のように書く。

```
for 変数 in 式1..式2
  命令1
  ⋮
  命令n
end
```

3.6　章末問題

練習 3.5 (条件分岐) 次のような計算を行う関数を定義せよ。

a) 2 次方程式 ax^2+bx+c $= 0$ の実数解の個数を求める `solutions(a,b,c)`.
判別式の値だけでなく、1 次方程式になっている場合も考えよ。

b) 2 次方程式 ax^2 + bx + c $= 0$ の実数解を 1 つ求める `solve1(a,b,c)`.
判別式の値だけでなく、1 次方程式になっている場合も考えよ。

c) 3 つの異なる値 x, y, z が与えられたときの中央値を求める `median(x,y,z)`.
中央値とは大きさ順に並べたときに真ん中に来る値のことである。

d) 年間所得が `income` 円だったときの所得税の税額を求める `income_tax(income)`.
ただし 2008 年の時点での日本の所得税率は次のようになっている。

(a) 195 万円以下の所得に対して 5%.

(b) 195 万円より多く 330 万円以下の所得に対して 10%.

(c) 330 万円より多く 695 万円以下の所得に対して 20%.

(d) 695 万円より多く 900 万円以下の所得に対して 23%.

(e) 900 万円より多く 1,800 万円以下の所得に対して 33%.

(f) 1,800 万円より多い所得に対して 40%.

ただし、上の税率は超過分についてのみ適用される。つまり、所得が 250 万円だった場合は、そのうちの 195 万円に対して 5%, 残りの55万円に対して10%の税が課されるので、$195 \times 0.05 + (250 - 195) \times 0.10 = 15.25$(万円) が税額となる。

東大生の家庭のうち約 19%は、950 万円から 1050 万円の年収を得ているという。収入と所得が同じだったとして、年間所得 1000 万円の場合の税額を計算してみよ。

学生生活実態調査 (2006 年・東京大学) の結果より。

e) 西暦 year 年 2 月の日数を求める days_of_february(year). ただし、year は次の条件を満たすときは閏年であることに注意せよ。

   (a) year が 4 の倍数である場合は閏年である

   (b) 条件 (a) を満たしていても、year が 100 の倍数の場合は閏年でない

   (c) 条件 (b) を満たしていても、year が 400 の倍数の場合は閏年である。

f) 西暦 year 年 month 月の日数を求める days_of_month(year,month).

練習 3.6 (条件式の組み合わせ) 練習 3.5c で定義した関数 median を定義し直して、(a)if y < x という形の条件式だけ使った median1 と、(b) && などを使って条件式を組み合て条件分岐の数を最小にした median2 を作れ。どちらの定義が分かりやすいだろうか。

練習 3.7 (真偽値を求める関数) 次のような計算を行う関数を定義せよ。

a) x が y で割り切れることを判定する divisible(x,y).

b) x < y < z であることを判定する ascending(x,y,z)

c) 西暦 y 年が閏年であるときに真となる leap_year(y).

   ただし現行の閏年の条件は「西暦年が 4 で割り切れる年を閏年とする。但し西暦年が 100 で割り切れるものの中でさらに 4 で (100 で割った) 商が割り切れない年は平年とする」というものである。従って 1900 年は平年、2000 年は閏年、2008 年も閏年、2100 年は平年である。

d) 添字 i が配列 a の範囲内かどうかを判定する within_range(a,i). ただし配列 a の添字は 0 からはじまり、a.length() の値未満である。

e) 座標 (x, y) が濃淡画像 img の中にあるかを判定する within_image(img,x,y).
   (ヒント: 上で定義した within_range が使える。)

練習 3.8 (論理関数) 真偽値を与えて真偽値を答えるような関数は**論理関数**という。次のような真理値表の通りに答えを返す論理関数 xor(x,y), implies(x,y) を定義せよ。

「情報」(川合編, 東京大学出版会)p. 158 参照。

x	y	xor(x,y)	implies(x,y)
false	false	false	true
false	true	true	true
true	false	true	false
true	true	false	true

練習 3.9 (文字列) 次のような関数を定義せよ。

a) 2 つの文字列 s, t のうち、長い方の文字列を求める longer(s,t).

b) 長さ 2 以上の文字列 s の先頭と最後の文字を取り除いた文字列を作る trim(s). 例えば trim("abc")⇒"b" となる。

c) 文字列 s の前半と後半を入れ替えた文字列を作る upsidedown(s). 例えば upsidedown("takeover")⇒"overtake" となる。余力があれば奇数文字のときに upsidedown("least")⇒"stale"となるように定義してみよ。

練習 3.10 (条件分岐を使った配列の操作)　a) a は 1 次元の配列とする。a の x 番目とその前後の要素数を数える関数 length3(a,x) を作れ。

$$\texttt{length3}(\texttt{a},\texttt{x}) = \begin{cases} 3 & (0 < \texttt{x} < \ell - 1) \\ 1 & (\texttt{x} = 0 \text{ かつ } \ell = 1) \\ 2 & (\ell > 1 \text{ かつ } (\texttt{x} = 0 \text{ または } \texttt{x} = \ell - 1)) \end{cases}$$

ただし ℓは配列 a の長さ

注: この関数は次に定義する sum3(a,x) で合計する要素の個数を求めている。

b) 練習 2.2 で定義した image_average9(image,x,y) および練習 2.3 で定義した sum3(a,x), array_average3(a,x) を、x(や y) の値が配列の端だった場合でも正しく計算できるように定義を修正した関数image_average(image,x,y), sum, array_average(a,x) を定義せよ。例えば a が [1,2,3,4] のとき sum3(a,3) は 3 と 4 の平均である 3.5 と答える。

c) 配列 a の i 番目と j 番目の要素を入れ替える関数 swap(a,i,j) を作れ。例えば a = [0,1,2,3]↵ swap(a,1,3) の順に命令を実行した場合、a は [0,3,2,1] となる。

d) 配列 a の i 番目と i + 1 番目の数値の大きさを比べ、前者が後者より大きいときに両者を入れ替える関数 swap_ascending(a,i) を作れ。

例えば a = [3,1,4,1]↵swap_ascending(a,0) の順に命令を実行した場合、a は [1,3,4,1] となる。さらにそこで swap_ascending(a,1) を実行しても a は [1,3,4,1] のままになる。

練習 3.11 (数列を作る) 大きさ n で i 番目の値が $i/\mathtt{n}$ になっているような 1 次元配列を作る関数 gradation(n) を定義せよ。(ヒント: make1d で 0 を代入している所を i を使った式に変える。整数どうしの割り算は整数商を求めてしまうことに注意せよ。)

練習 3.12 (色々な図形 *) 図 3.2 のような画像を作成する関数をそれぞれ定義せよ。

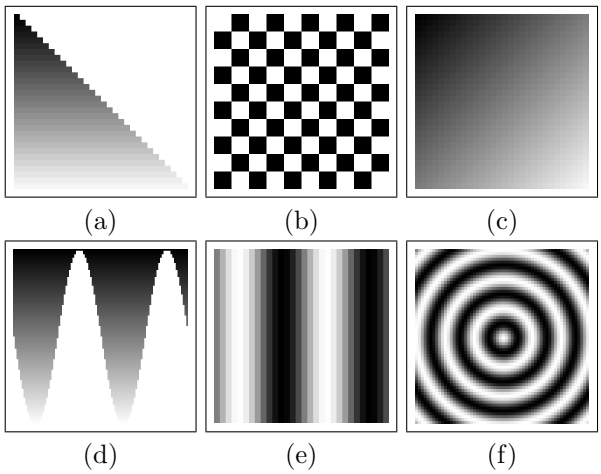


図 3.2: 繰り返しを使って作った色々な画像

練習 3.13 (画像の加工 **) 画像を作成する場合に、他の画像の情報を参照することで画像を加工することができる。例えば、濃淡画像 s を明るくするには、例えば各点の明度 b を $\frac{b+1}{2}$ にしてやればよいので、同じ大きさの画像 img を作り、各座標 (x, y) について以下のように定めてやればよい。

```
img[y][x] = (s[y][x]+1)*0.5
```

(配列の大きさは、s.length() で高さが、s[0].length() で幅が求められる。)

以下のような関数を定義し、練習 3.12 で作った画像に適用してみよ。

a) 与えられた画像 img を明るくした画像を作る関数 brighter(img).

b) 与えられた画像 img1, img2 を合成した画像を作る blend(img1,img2). ただし合成は、同じ座標の点の明度を平均化するものとする。

c) 与えられた画像 img を平均化する blur(img). (練習 2.2 で作った image_average9 を使う。)

配布プログラム utlogo.rb には、東京大学のロゴマークの白黒濃淡画像の配列を作る関数 logo() が定義されている。例えば上で定義した blur(img)

を使って `show(blur(logo()))`, `show(blur(blur(logo())))` とすると、次のようにぼやけた画像を作ることができる。

`logo()`

`blur(logo())`

`blur(blur(logo()))`

第4章　関数から計算へ

これまでの章では、画像の作成を除けば、単純な関数の組み合わによって複雑な計算を行う方法だけを見てきた。コンピュータの能力を引き出すためには、単純な関数を**何度も行って**答えを得る**反復**が必要になる。

この章では反復を行わせる 2 つの方法と、その応用を紹介する。第 4.1 節では**繰り返し**という、コンピュータに直接的に反復動作を行わせる方法を紹介する。第 4.2 節では**再帰**という、関数の定義に自分自身を使う方法によって反復を表わす方法を紹介する。第 4.3 節では配列や文字列を反復によってに扱う例をいくつか紹介する。

この章は繰り返しを先に再帰を後に紹介をしているが、第 4.1 節と第 4.2 節のどちらから先に読んでも構わないように書かれている。繰り返しによる定義はコンピュータ内部動作を考えやすい、再帰による定義はその計算が意味することを把握しやすいという特徴があるので両方を知っておくことが重要である。

4.1　繰り返しによる反復計算の定義

コンピュータの特徴の 1 つは、同じ計算を高速に反復して行う能力である。すでに第 3.4 節では、明度の計算を画像の各点に対して行うという簡単な反復処理の方法を見ている。

この反復処理は画像の作成だけでなく、例えば「1 から 100 までの数の和」や「100 番目の素数」のような値を求めるためにも使われる。

そのためには、反復して何かを計算しつつ、その結果を集めることが必要になる。まずは変数を使って結果を集めてゆく方法を見て、それを反復の中で使う方法を見てゆこう。

4.1.1　代入による変数の更新

第 1.3.2 節では、`weight=104.0` として BMI を計算した後、`weight=104.0*0.9` を実行して体重が 1 割減った場合の BMI を計算する例を見た。

このような代入で、右辺に同じ変数を使うこともできる。

```
1 irb(main):003:0> weight=104.0
2 => 104.0
```

```
3 irb(main):004:0> weight=weight-10
4 => 94.0
5 irb(main):005:0> weight
6 => 94.0
7 irb(main):006:0> weight=weight-10
8 => 84.0
```

3 行目の意味は、「現在の weight から 10 を引き、その結果を weight に代入する」というものである。別の言い方をすると、この命令は「weight の値を 10 減らす」ものだと言える。

当然、同じ代入命令をもう一度実行すると (7 行目)、そのときの weight の値から 10 引いた値に変更されるので、新しい weight の値はさらに小さなものになる。

4.1.2　繰り返しによる和の定義

まず、1 から n までの和を求める関数を考える。この計算は合計を表わす変数 s を用意して、「s を i だけ増やす」ことを $i \in 1 \ldots n$ について行えばよい。第 3.4.2 節で見たように for 命令を使うことで、次のように定義することができる。

```
1 # sum of the numbers from 1 to n
2 def sum_loop(n)
3     s = 0
4     for i in 1..n
5         s = s+i
6     end
7     s
8 end
```

ファイル 4.1: sum_loop

なおこの定義では、繰り返しによって書かれた関数を、再帰によって定義したものと区別するため、関数名に_loop を加えている。

(章末の練習問題: 4.6)

4.1.3　条件を満たす値を探す繰り返し

「1 から n までの数のうち、最大の k の約数 $\mathrm{gd}(k, n)$」を求める計算を繰り返しによって行うにはどうすればよいだろうか。再帰的関数を使った計算

(第 4.2.3 節) では、$n,\ n-1,\ n-2,\ \ldots$ の順に k の約数かどうかを調べ、最初に見つけた約数を答えとしていた。

これと同じ計算を繰り返しによって行うためには、新しい命令が必要になる。いままで使っていた for は「1 から n まで」のように回数を先に決めて繰り返す命令であるため、条件を満たす値が見つけるような場合は、何回繰り返せばよいかは分からないためである。

Ruby に用意されている、もう 1 つの繰り返しの命令は「ある条件が成り立っている間は計算を続ける」というものである。これを使って $\mathrm{gd}(k,n)$ を定義すると次のようになる。

```
1 # the greatest divisor of k in between 1 to n
2 load("./divisible.rb")
3
4 def gd_loop(k,n)
5     while !divisible(k,n)
6         n = n-1
7     end
8     n
9 end
```

ファイル 4.2: gd_loop

5 行目から 7 行目の while から end までが繰り返しの命令であり、while の後ろの式が繰り返しを続ける条件、次の行から end までが繰り返される命令である。従ってこの場合は「k が n で割り切れない間、n を 1 減らす計算を続ける」という意味である。例えば gd_loop(6,5) であれば

- k = 6, n = 5 のときに k が n で割り切れないので、
- n を 1 減らし、
- k = 6, n = 4 のときに k が n で割り切れないので、
- n を 1 減らし、
- k = 6, n = 3 のときに k が n で割り切れるので繰り返しを終える。

とい繰り返しになる。その後、8 行目でその時の—つまり k の約数になっていた—n の値を答えとするので、最初に見つけた約数を求めたことになる。

(章末の練習問題: 4.8)

4.2　再帰による反復計算の定義

コンピュータの特徴の 1 つは、同じ計算を高速に反復して行う能力である。すでに第 3.4 節では、明度の計算を画像の各点に対して行うという簡単な反復処理の方法を見ている。

この反復処理は画像の作成だけでなく、例えば「1 から 100 までの数の和」や「100 番目の素数」のような値を求めるためにも使われる。

このような計算を定義するための一つの方法は**再帰**である。以下では簡単な数学関数を例にとって、再帰的な関数の定義の方法を説明する。

4.2.1　再帰による和の定義

例として「1 から n までの和を求める関数 $\mathrm{sum}(n)$」を定義することを考えよう。式で書くと $\mathrm{sum}(n)$ は次のようになる。

$$\mathrm{sum}(n) = 1 + 2 + \cdots + n \tag{4.1}$$

この式は、そのままでは Ruby の関数にはならない。なぜなら、与えられた n によって式が変わってしまうからである。

ではどうすればよいだろうか。ここで $\mathrm{sum}(n-1)$ がどのような式かを考えてみよう。$n \geq 2$ の場合は次のようになる。

$$\mathrm{sum}(n-1) = 1 + 2 + \cdots + (n-1) \quad (\text{ただし } n \geq 2) \tag{4.2}$$

また、

$$\mathrm{sum}(n) = 1 \quad (n = 1 \text{ の場合}) \tag{4.3}$$

である。

式 4.1 と式 4.2 から、式 4.1 の右辺は $\mathrm{sum}(n-1) + n$ と表わせる (ただし $n \geq 2$ の場合) ことに気付く。$n = 1$ の場合とあわせると $\mathrm{sum}(n)$ には次のような関係があると言える。

$$\mathrm{sum}(n) = \begin{cases} \mathrm{sum}(n-1) + n & (n \geq 2) \\ 1 & (n = 1) \end{cases} \tag{4.4}$$

式 4.4 は、sum という関数を表わすために再び sum を使っている。この関係を左辺から右辺に展開してゆくことで和が求められる。実際に $\mathrm{sum}(3)$ のような式を展開することで確認してみよう。式の展開を $\mathrm{sum}(3) \Rightarrow \mathrm{sum}(2) + 3$ のように書くことにすると、

$$\mathrm{sum}(3) \Rightarrow \mathrm{sum}(2) + 3 \Rightarrow (\mathrm{sum}(1) + 2) + 3 \Rightarrow (1 + 2) + 3$$

となり、確かに 1 から 3 の和になっていることが分かる。

```
1 # sum of the numbers from 1 to n
2 def sum(n)
3     if n >= 2
4         sum(n-1) + n
5     else
6         1
7     end
8 end
```

ファイル 4.3: sum.rb

式 4.4 のように書かれた関係は、そのまま Ruby の関数として定義することができる。場合分けは条件分岐を使えばよいので、次のような定義になる。

これまでの関数定義と唯一違うのは、関数 sum の定義の中 (4 行目) に、関数 sum 自身を使った式が表われる点である。このような定義のことを、**再帰的関数**という。

実際に計算させてみると、確かに和を求めていることが分かる。

```
1 irb(main):004:0> sum(3)
2 => 6
3 irb(main):005:0> sum(10)
4 => 55
5 irb(main):006:0> 1+2+3+4+5+6+7+8+9+10
6 => 55
```

ここまでで見たことをまとめると次のように言える。

> **n 回の反復によって値を求めるような計算 $f(n)$ は、$f(n)$ と $f(n-1)$ の関係を見つければ、それを Ruby の再帰的関数として定義することができる。**

(章末の練習問題: 4.9 4.10)

4.2.2　約数の和

次に「1 から n までの数のうち、k の約数になっているものの和 $\mathrm{sod}(k,n)$」を定義することを考えてみよう。前に定義した sum との違いは下線部分、つまり k の約数だけを選んで和を求めるという点だけであるので、$\mathrm{sod}(k,n)$ と

関数名 sod は sum of divisors (約数の和) の略。

$\mathrm{sod}(k, n-1)$ の関係は次のように表わせる。

$$\mathrm{sod}(k,n) = \begin{cases} \mathrm{sod}(k,n-1)+n & (n \geq 2 \text{ かつ } n \text{ が } k \text{ の約数}) \\ \mathrm{sod}(k,n-1)+0 & (n \geq 2 \text{ かつ } n \text{ が } k \text{ の約数でない}) \\ 1 & (n = 1) \end{cases} \tag{4.5}$$

式 4.4 と見比べると、n が k の約数でない場合には 0 を足している点だけが違う。これを Ruby の関数として表わすと以下のようになる。

```
1  # sum of divisors of k in between 1 and n
2  load("./divisible.rb")
3
4  def sod(k,n)
5      if n >= 2
6          if divisible(k,n)
7              sod(k,n-1)+n
8          else
9              sod(k,n-1)
10         end
11     else
12         1
13     end
14 end
```

ファイル 4.4: `sod.rb` (ここでは 練習 3.7a で定義した `divisible(k,n)` を使って `n` が `k` の約数かどうかを判定している。)

```
1  irb(main):004:0> sod(10,9)
2  => 8
3  irb(main):005:0> 5+2+1
4  => 8
5  irb(main):006:0> sod(28,27)
6  => 28
7  irb(main):007:0> 14+7+4+2+1
8  => 28
```

練習 4.1 (素数の判定) 素数とは、1 と自分自身しか約数がないような数である。上で定義した関数 `sod` を使って 2 以上の整数 `n` が素数のときにのみ `true`, そうでないときに `false` となるような関数 `prime(n)` を定義せよ。(`n` が 1 の場合は考えなくてよい。つまり、`prime(1)` の答は `true` でも `false` でもよいとする。)

```
irb(main):007:0> prime(6)
=> false
irb(main):008:0> prime(7)
=> true
irb(main):009:0> prime(107)
=> true
irb(main):010:0> prime(961)
=> false
```

練習 4.2 (組み合わせ数) n 個から k 個を選ぶ組み合わせ数 ${}_nC_k$ を求める `combination(n,k)` を定義せよ。ただし ${}_nC_k$ は、次のような関係を満たしている。

$$ {}_nC_k = \begin{cases} 0 & (k > n \text{ のとき}) \\ 1 & (k = 0 \text{ のとき}) \\ {}_{n-1}C_{k-1} + {}_{n-1}C_k & (\text{それ以外}) \end{cases} \tag{4.6} $$

この関係は「n 個から k 個を選ぶ」方法は、n 個の中の最初の 1 個に注目すると「それを選び、残りの $n-1$ 個から $k-1$ 個を選ぶ」か「それを選ばず、残りの $n-1$ 個から k 個を選ぶ」に限られるからだと理解できる。

(章末の練習問題: 4.11)

4.2.3　条件を満たす値を探す

「1 から n までの数のうち、最大の k の約数」を求める関数 $\mathrm{gd}(k,n)$ を考えてみよう。前に定義した sod からの類推で、n が k の約数の場合と、そうない場合について考えてみる。もし n が k の約数だったとすると、明らかに $\mathrm{gd}(k,n) = n$ である。また n が k の約数でないとすると、最大の約数は $n-1$ 以下なので $\mathrm{gd}(k,n) = \mathrm{gd}(k,n-1)$ となるはずである。まとめると次のような関係になる。

関数名 gd は greatest divisor(最大の約数) の略。

$$ \mathrm{gd}(k,n) = \begin{cases} n & (n \text{ が } k \text{ の約数}) \\ \mathrm{gd}(k,n-1) & (n \text{ が } k \text{ の約数でない}) \end{cases} \tag{4.7} $$

(この関係には $n = 1$ の場合がないように見えるが、1 は常に k の約数になるので 1 つ目の場合に含まれている。)

これを Ruby の関数にするとファイル 4.5 のようになる。このようにして定義された `gd` の計算は n を 1 つずつ減らしていって最初に k の約数となった時に終わる。例えば `gd(6,5)` の場合であれば

$$ \mathtt{gd}(6,5) \overset{5 \text{ は } 6 \text{ の約数でない}}{\Longrightarrow} \mathtt{gd}(6,4) \overset{4 \text{ は } 6 \text{ の約数でない}}{\Longrightarrow} \mathtt{gd}(6,3) \overset{3 \text{ は } 6 \text{ の約数}}{\Longrightarrow} \mathtt{3} $$

のような順で展開され、答えの 3 を得る。

```
# the greatest divisor of k in between 1 to n
load("./divisible.rb")

def gd(k,n)
    if divisible(k,n)
        n
    else
        gd(k,n-1)
    end
end
```

ファイル 4.5: gd.rb

練習 4.3 (素数の判定の改良) 素数とは、自身を除く最大の約数が 1 であるような数だとも言える。関数 gd を使って素数の判定をする関数 prime2(n) を定義せよ。練習 4.1 で定義した prime と計算回数の違いがあるかを考えよ。

(章末の練習問題: 4.12)

4.3　配列・文字列と繰り返し

繰り返しを使った計算は、配列や文字列のように、似たような値が沢山並んでいるデータを扱う場合に威力を発揮する。すでに前章では画像を作成する例を見たが、ここでは違う種類の例として、組み合わせ数の計算と、文字列の探索を紹介する。

4.3.1　配列と繰り返しを使った組み合わせ数の計算

練習 4.2 に出てきた組み合わせ数 ${}_nC_k$ の計算は、そのまま繰り返しにすることは難しい。これは、組み合わせ数が単純に値を積み上げて決まるような性質がないからである。

このような場合でも、配列と繰り返しを使うと上手く計算ができることがある。まずは組み合わせ数 ${}_nC_k$ を表にしてみよう (表 4.1)。

この表の右上半分は $k > n$ なのですべて 0 であるが、組み合わせ数としては意味がないので空欄にしている。

この表は上の行から順に埋めてゆくことができる。各行について、まず左端 ($k = 0$) と対角線上 ($k = n$ となる所) はすべて 1 である。それ以外の部分は、${}_nC_k = {}_{n-1}C_{k-1} + {}_{n-1}C_k$ なので、真上と左上にある 2 つの数の和を求めてやればよい。

${}_nC_n = 1$ であることは式 4.6(p. 52) の定義から簡単に証明できる。

この表を、配列と繰り返しによって作ってみよう。以下は表を作り、そこから ${}_nC_k$ の値をとり出すような関数 combination_loop(n,k) の定義であ

n \ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

表 4.1: 組み合わせ数 ${}_nC_k$ の値

る。この関数はまず表をあらわす配列として大きさ (n+1)×(n+1) の配列を作る (4 行目)。そして配列の 0 行目から n 行目について、左端 (6 行目) と対角線上 (10 行目) を 1 にして、その間を内側の繰り返しによって埋めている。8 行目が真上と左上にある数を足した値をしまっているところである。このようにして表が完成したら、表の n 行 k 列目が求める答である (12 行目)。

```
1  load("./make2d.rb")
2
3  def combination_loop(n,k)
4    c=make2d(n+1,n+1)
5    for i in 0..n
6      c[i][0] = 1
7      for j in 1..(i-1)
8        c[i][j] = c[i-1][j-1] + c[i-1][j]
9      end
10     c[i][i] = 1
11   end
12   c[n][k]
13 end
```

ファイル 4.6: combination_loop

練習 4.4 (再帰と繰り返しの比較) 練習 4.2 で定義した combination とここで定義した combination_loop が同じ結果を返すことを、いくつかの n, k について確めよ。また、n, k を大きくした場合にはどちらが速いか、実際に計算させて比べてみよ。

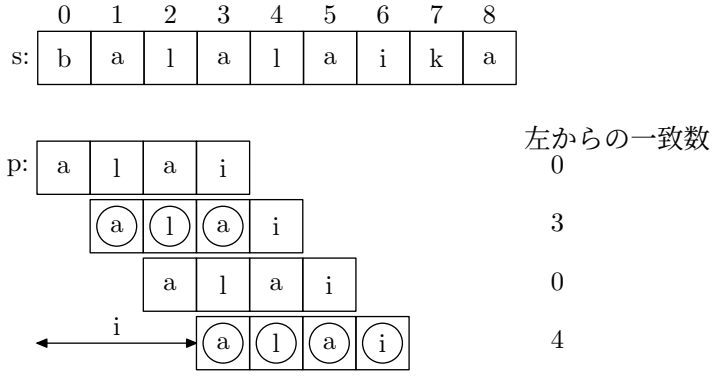
(章末の練習問題: 4.13 4.14)

4.3.2　言葉探し

文字列に関する処理には、長い文字列の中に特定の文字列が含まれている場所を見つけるというものがある。この処理は、ワードプロセッサの検索機能や、DNA の塩基配列の解析など多くの場面で使われる。

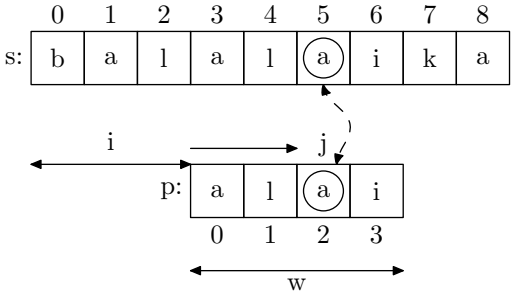
例として「balalaika」という文字列 (s とする) の中に「alai」という文字列 (p とする) が何文字目に現われるかを探す問題を考えよう。話を簡単にするために、s のどこかに必ず p が現われることは分かっているものとする。また 2ヶ所以上に現われる場合には、一番左側の場所を答えることにする。

下の図を使って探し方を説明する。



文字列 s と p を縦に並べて、左から何文字目まで一致しているかを調べる。図では丸の付いた文字が一致を表わし、右端にその個数を書いている。一致数が p の長さに満たない場合は p を 1 文字ずつ右にずらして繰り返す。そうすると一致数と p の長さが等しくなったときのずらした文字数が求める値になる。

このやり方では、s の i 文字目からと p が何文字一致しているかを調べる必要がある。それを関数 `submatch(s,i,p,w)` として定義してみよう。ただし、w は p の長さである。



関数 submatch は、上図のように j を $0, 1, 2, \ldots$ と変えながら s の i + j 文字目と p の j 文字目を比べてゆく。両者が一致しなくなったときは 0 から j − 1 文字目までは一致していたので一致数は j である。また、j が w になっ

たときは 0 から w − 1 文字目まですべて一致していたことになるので一致数はやはり j = w である。

従って j が w より小さくかつ、s の i + j 文字目と p の j 文字目が等しいあいだは繰り返しを続けて、どちらかの条件が成り立たなくなったときの j の値が一致数となる。Ruby の関数としては次のように定義できる。

```
1  def submatch(s,i,p,w)
2    j = 0
3    while j < w && s[(i+j)..(i+j)] == p[j..j]
4      j = j + 1
5    end
6    j
7  end
```

ファイル 4.7: match.rb (前半)

3 行目の後半の条件は、s の i+j 文字目と p の j 文字目が等しいかどうかを調べている。第 3.3 節で説明したように、Ruby では文字列 t の k 文字目は t[k..k] という式によって取り出すことに注意せよ。

例えば submatch("balalaika", 1, "alai", 4) の値は 3, submatch("balalaika", 3, "alai", 4) の値は 4 になる。

関数 match は、i を $0, 1, 2, \ldots$ と変えながら、submatch(s, i, p, w) が w に等しくなるような最小の i を求めればよいので、次のようになる。

```
8  def match(s,p)
9    i=0
10   w=p.length()
11   while submatch(s,i,p,w) < w
12     i = i + 1
13   end
14   i
15 end
```

ファイル 4.8: match.rb (後半)

なお、10 行目の p.length() は文字列の p の長さを求める式である (第 3.3 節参照)。

練習 4.5 (RNA 塩基配列の探索) 配布プログラム rna_sample.rb にある関数 seq0(), seq1() は RNA 塩基配列の例である。これらの中に AUG という文字列が表われる場所を探せ。

(章末の練習問題: 4.15)

4.4　定義のまとめ

条件を満たすまでの繰り返し: 式 が成り立つ間 命令$_1$ から 命令$_n$ を毎回実行する**繰り返し** `while` は次のように書く。

```
while 式
  命令1
  ⋮
  命令n
end
```

この命令は、まず (☆) 式 の値を求め、それが真であれば 命令$_1$ から 命令$_n$ までを順に実行し、再び☆からの手順を繰り返す。式 の値が偽になったら `end` 以降の命令に移る。

4.5　章末問題

練習 4.6 (繰り返しによって値を集める) 繰り返しによって以下のような値を求める Ruby の関数を定義せよ。

a) `n` の階乗、即ち 1 から `n` までの積 `n!` を求める `factorial_loop(n)`.

b) $2^{\mathtt{n}}$ を求める `power2_loop(n)`. 記号`**`を使わずに定義せよ。

c) $\mathtt{x}^{\mathtt{n}}$ を求める `power_loop(x,n)`. 同じく記号`**`を使わずに定義せよ。

d) 次の式に示されるような級数を求める `taylor_e_loop(x,n)`.

$$\sum_{k=0}^{\mathtt{n}} \frac{\mathtt{x}^k}{k!}$$

($\mathtt{n} \to \infty$ のとき、この式は $e^{\mathtt{x}}$ の Taylor 級数になっている。)

練習 4.7 (繰り返しによって条件を満たす値を集める) 繰り返しによって以下のような値を求める Ruby の関数を定義せよ。

a) 1 から `n` までの数のうち、`k` の約数になっているものの 個数(number of divisors) を求める `nod_loop(k,n)`. (ヒント: 式 4.5 の足す数を変える。)

b) 1 から n までの数のうち、素数になっているものの個数 (number of primes) を求める nop_loop(n).

c) 1 から n までの数について、それぞれ自身を含むような約数の和を求めたとき、最大のもの (maximum sum of divisors) を求める msod_loop(n).

練習 4.8 (繰り返しによって条件を満たす値を探す) 繰り返しによって以下のような値を求める Ruby の関数を定義せよ。

a) n より大きくかつ最小の素数—つまり次の素数 (next prime number) —を求める np_loop(n).

b) 素数 p よりも大きな素数のうち n 番目の素数を求める nth_prime_loop(p,n).

c) 第 3.2 節の式 3.1(p. 34) で紹介した関数 tnpo(n) は n が偶数なら 1/2, 奇数なら 3 倍して 1 加えた数を求めるものだった。数学者 Collatz はどんな整数 n が与えられたときでも、この関数を使って数を変化させてゆくと、いずれ 1 になると予想した。例えば 3 から始めた場合は $3 \Rightarrow 10 \Rightarrow 5 \Rightarrow 16 \Rightarrow 8 \Rightarrow 4 \Rightarrow 2 \Rightarrow 1$ といった具合に予想通りになっていることが確められる。

この予想は、反例は見つかっていないが証明もされていない、数学の未解決問題の 1 つである。

そこで n から上の手順で数を変化させて 1 になるまでの回数を collatz(n) とする。例えば collatz(5) = 5, collatz(16) = 4 である。

collatz(n) を求める Ruby の関数 collatz_loop(n) を定義せよ。

d) n 以上の整数で最小の完全数を見つける next_perfect_loop(n). ただし完全数 k とは k を除く約数の和が k と等しいような数のことである。例えば 6 の約数は (6 を除くと) 3, 2, 1 だけであり、その和が 6 になるので 6 は完全数である。最初に n が完全数であるかを判定する関数 perfect_loop(n) を定義しておくとよい。

練習 4.9 (再帰的に値を集める) 以下のような値を求める再帰的な Ruby の関数を定義せよ。繰り返しは使わないこと。

a) n の階乗、即ち 1 から n までの積 $n!$ を求める factorial(n).

b) 2^n を求める power2(n). 記号**を使わずに定義せよ。(ヒント: 2^n は、$n \geq 1$ の場合に $2^n = 2 \times 2^{n-1}$ を満たしている。)

c) x^n を求める power(x,n). 同じく記号**を使わずに定義せよ。

d) 次の式に示されるような級数を求める taylor_e(x,n).

$$\sum_{k=0}^{n} \frac{x^k}{k!}$$

($n \to \infty$ のとき、この式は e^x の Taylor 級数になっている。)

練習 4.10 (文字列を作る再帰) ファイル 4.3 の 6 行目の数値 `1` を文字列 `"1"` に置きかえた場合、`sum(n)` はどのような答えを返すか?

練習 4.11 (再帰的に条件を満たす値を集める) 以下のような値を求める再帰的な Ruby の関数を定義せよ。繰り返しは使わないこと。

a) 1 から `n` までの数のうち、`k` の約数になっているものの 個数(number of divisors) を求める `nod(k,n)`. (ヒント: 式 4.5 の足す数を変える。)

b) 1 から `n` までの数のうち、素数になっているものの個数 (number of primes) を求める `nop(n)`.

c) 1 から `n` までの数について、それぞれ自身を含むような約数の和を求めたとき、最大のもの (maximum sum of divisors) を求める `msod(n)`.

$\mathrm{sod}(k,k) = s_k$ と書くことにすると、この問題は s_1 から s_n の最大値を求めることに等しい。「s_1 から s_n の最大値」は「s_1 から s_{n-1} の最大値」より s_n が大きい場合とそうでない場合に分けて考えればよいので、次のような関係を満たしている。

$$\texttt{msod}(n) = \begin{cases} \texttt{msod}(n-1) & (\texttt{msod}(n-1) \geq s_n) \\ s_n & (\texttt{msod}(n-1) < s_n) \\ s_1 & (n = 1) \end{cases}$$

練習 4.12 (再帰的に条件を満たす値を探す) 以下のような値を求める再帰的な Ruby の関数を定義せよ。繰り返しは使わないこと。

a) `n` より大きくかつ最小の素数—つまり次の素数 (next prime number) —を求める `np(n)`. まず `n` が素数だった場合は `np(n)` = `n` である。そうでない場合は、`np(n)` は「`n`＋1 より大きくかつ最小の素数」に一致する。

b) 素数 `p` よりも大きな素数のうち n 番目の素数を求める `nth_prime(p,n)`. 例えば `nth_prime(5,3)` は 5 よりも大きな素数のうち 3 番目であり、5 の次の素数は 7 なので 7 よりも大きな素数のうち 2 番目つまり `nth_prime(7,2)` と等しい。これはさらに 7 の次の素数 11 よりも大きな素数のうち 1 番目と等しいので 11 の次の素数 13 が答になるはずである。

c) 第 3.2 節の式 3.1(p. 34) で紹介した関数 $\mathrm{tnpo}(n)$ は n が偶数なら 1/2, 奇数なら 3 倍して 1 加えた数を求めるものだった。数学者 Collatz はどんな整数 n が与えられたときでも、この関数を使って数を変化させてゆくと、いずれ 1 になると予想した。例えば 3 から始めた場合は $3 \Rightarrow 10 \Rightarrow 5 \Rightarrow 16 \Rightarrow 8 \Rightarrow 4 \Rightarrow 2 \Rightarrow 1$ といった具合に予想通りになっていることが確められる。

この予想は、反例は見つかっていないが証明もされていない、数学の未解決問題の 1 つである。

そこで n から上の手順で数を変化させて 1 になるまでの回数を $\mathrm{collatz}(n)$ とする。例えば $\mathrm{collatz}(5) = 5$, $\mathrm{collatz}(16) = 4$ である。

(a) collatz(n) と collatz(tnpo(n)) の関係を書け。

(b) collatz(n) を求める Ruby の関数 `collatz(n)` を定義せよ。

d) `n` 以上の整数で最小の完全数を見つける `next_perfect(n)`. ただし完全数 k とは k を除く約数の和が k と等しいような数のことである。例えば 6 の約数は (6 を除くと) 3, 2, 1 だけであり、その和が 6 になるので 6 は完全数である。最初に `n` が完全数であるかを判定する関数 `perfect(n)` を定義しておくとよい。

練習 4.13 (Eratosthenes の篩 *) 素数を求める方法の 1 つに Eratosthenes の篩(ふるい) というものがある。考え方はとしては「整数が素数かどうか」を記録する大きな表を用意して、素数を発見するたびにその倍数を表から消してゆくものである。

まず 2 から始まる整数の列を用意する。

2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,
52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,
77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,

列の先頭の数 2 はまだ消されていないので素数である。2 に丸を付け、2 の倍数を消す。

②, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9,~~10~~,11,~~12~~,13,~~14~~,15,~~16~~,17,~~18~~,19,~~20~~,21,~~22~~,23,~~24~~,25,~~26~~,
27,~~28~~,29,~~30~~,31,~~32~~,33,~~34~~,35,~~36~~,37,~~38~~,39,~~40~~,41,~~42~~,43,~~44~~,45,~~46~~,47,~~48~~,49,~~50~~,51,
~~52~~,53,~~54~~,55,~~56~~,57,~~58~~,59,~~60~~,61,~~62~~,63,~~64~~,65,~~66~~,67,~~68~~,69,~~70~~,71,~~72~~,73,~~74~~,75,~~76~~,
77,~~78~~,79,~~80~~,81,~~82~~,83,~~84~~,85,~~86~~,87,~~88~~,89,~~90~~,91,~~92~~,93,~~94~~,95,~~96~~,97,~~98~~,99,

次に消されていない数 3 も素数である。3 に丸を付け、3 の倍数を消す。

②,③, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~,~~10~~,11,~~12~~,13,~~14~~,~~15~~,~~16~~,17,~~18~~,19,~~20~~,~~21~~,~~22~~,23,~~24~~,25,~~26~~,
~~27~~,~~28~~,29,~~30~~,31,~~32~~,~~33~~,~~34~~,35,~~36~~,37,~~38~~,~~39~~,~~40~~,41,~~42~~,43,~~44~~,~~45~~,~~46~~,47,~~48~~,49,~~50~~,~~51~~,
~~52~~,53,~~54~~,55,~~56~~,~~57~~,~~58~~,59,~~60~~,61,~~62~~,~~63~~,~~64~~,65,~~66~~,67,~~68~~,~~69~~,~~70~~,71,~~72~~,73,~~74~~,~~75~~,~~76~~,
77,~~78~~,79,~~80~~,~~81~~,~~82~~,83,~~84~~,85,~~86~~,~~87~~,~~88~~,89,~~90~~,91,~~92~~,~~93~~,~~94~~,95,~~96~~,97,~~98~~,~~99~~,

次の数 4 は消されているので素数でない。その次の 5 が素数であるので同様に丸を付け倍数を消す。

②,③, ~~4~~,⑤, ~~6~~, 7, ~~8~~, ~~9~~,~~10~~,11,~~12~~,13,~~14~~,~~15~~,~~16~~,17,~~18~~,19,~~20~~,~~21~~,~~22~~,23,~~24~~,~~25~~,~~26~~,
~~27~~,~~28~~,29,~~30~~,31,~~32~~,~~33~~,~~34~~,~~35~~,~~36~~,37,~~38~~,~~39~~,~~40~~,41,~~42~~,43,~~44~~,~~45~~,~~46~~,47,~~48~~,49,~~50~~,~~51~~,
~~52~~,53,~~54~~,~~55~~,~~56~~,~~57~~,~~58~~,59,~~60~~,61,~~62~~,~~63~~,~~64~~,~~65~~,~~66~~,67,~~68~~,~~69~~,~~70~~,71,~~72~~,73,~~74~~,~~75~~,~~76~~,
77,~~78~~,79,~~80~~,~~81~~,~~82~~,83,~~84~~,~~85~~,~~86~~,~~87~~,~~88~~,89,~~90~~,91,~~92~~,~~93~~,~~94~~,~~95~~,~~96~~,97,~~98~~,~~99~~,

これを続けてゆくとやがて消されていない数が素数として残る。

②,③, ~~4~~,⑤, ~~6~~,⑦, ~~8~~, ~~9~~,~~10~~,⑪,~~12~~,⑬,~~14~~,~~15~~,~~16~~,⑰,~~18~~,⑲,~~20~~,~~21~~,~~22~~,㉓,~~24~~,~~25~~,~~26~~,
~~27~~,~~28~~,㉙,~~30~~,㉛,~~32~~,~~33~~,~~34~~,~~35~~,~~36~~,㊲,~~38~~,~~39~~,~~40~~,㊶,~~42~~,㊸,~~44~~,~~45~~,~~46~~,㊼,~~48~~,~~49~~,~~50~~,~~51~~,
~~52~~,㊿,~~54~~,~~55~~,~~56~~,~~57~~,~~58~~,~~59~~,~~60~~,61,~~62~~,~~63~~,~~64~~,~~65~~,~~66~~,67,~~68~~,~~69~~,~~70~~,71,~~72~~,73,~~74~~,~~75~~,~~76~~,
~~77~~,~~78~~,79,~~80~~,~~81~~,~~82~~,83,~~84~~,~~85~~,~~86~~,~~87~~,~~88~~,89,~~90~~,~~91~~,~~92~~,~~93~~,~~94~~,~~95~~,~~96~~,97,~~98~~,~~99~~,

この考え方をもとに、2 から `n` までの数が素数かどうかを示す配列を作る `primes(n)` を定義せよ。ただし、作られる配列の i 番目は、i が素数のとき 0, 素数でないとき 1 だとする。

練習 4.14 (Sierpinski の三角形) 大きさ `n` × `n` で、i 行 j 列目が ($_iC_j$ を 2 で割った余り) になっているような配列を作る関数 `sierpinski(n)` を定義せよ。この関数が作る配列を `show` を使って見ると図 4.1 のようになる (見易さ

のために白黒を逆にしている)。この図形は Sierpiński の三角形として知られているものである。

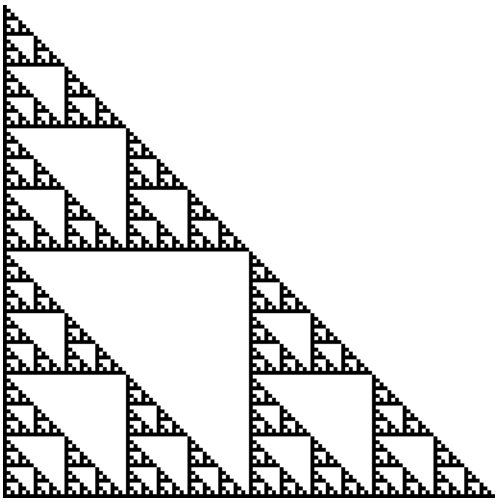


図 4.1: Sierpiński の三角形

練習 4.15 (見つからない場合) ファイル 4.8 の `match` の定義では `s` 中に `p` が必ず現われることを仮定していた。`p` が現われない場合に -1 と答える `match_safe(s,p)` を定義せよ。

第II部

プログラミングを通して学ぶ
情報科学の諸概念

第5章　アルゴリズムと計算量

コンピュータを使って問題解決をする場合には、解き方をいくつか考えて適切なものを選び、その解き方の通りに計算をするプログラムを作って実行する、という2つの段階を踏む。

アルゴリズムとはこの「解き方」のことである。より厳密には、問題を解くための手順で、有限の時間で答を出すことが保証されているものを指す。

問題が与えられたときに、いきなりプログラムを書かずに、アルゴリズムという中間段階を経由することには次のような意味がある。1つはプログラムを作る前から計算時間などがある程度—例えば「このやり方では一生かかっても終わらない」といった程度に—見積ることができる点である。もう1つはアルゴリズムに関する知識を問題から独立して蓄積できる点である。例えば、通信ネットワーク上に流すことができるデータ量を計算する問題と、工場においてどの機械にどの製品の製造させると効率的かという2つの問題は、問題としては全く違うもののように見えるが、実は共通のアルゴリズムを使って解くことができる。アルゴリズムを考えることで、こうした異なる問題についての知識やプログラムを使い回すことができるのである。

この章では、具体的な問題として Fibonacci 数列の計算を例にとる。まずこの問題を解くアルゴリズムが複数あることを見て、アルゴリズムによる計算時間の違いから、計算量の考え方を紹介する。さらに別の問題として整列のアルゴリズムを2つ紹介する。

5.1　Fibonacci 数を求めるアルゴリズム

ねずみ講、ねずみ算のようにネズミの増え方に関する話はよく知られている。ここではウサギの増え方を考てみよう。子ウサギは生後1ヶ月経つと親ウサギになり、以降、毎月1羽の子ウサギを生むようになるとする。このような場合に、

0月、1 羽の子ウサギがいたとする。	子
1月、親ウサギになる。	親
2月、1 羽の子ウサギを生む。	親子
3月、親ウサギはもう 1 羽の子ウサギを生み、子ウサギは親ウサギになる。	親親子
4月、2 羽の親ウサギがそれぞれ子ウサギを生み、子ウサギは親ウサギになる。	親親親子子
5月、3 羽の親ウサギがそれぞれ子ウサギを生み、2 羽の子ウサギは親ウサギになる。	親親親親親子子子
6月、5 羽の親ウサギがそれぞれ子ウサギを生み、3 羽の子ウサギは親ウサギになる。	親親親親親親親子子子子子

のように増えてゆくウサギの親子をあわせた総数を並べてみると

$$1,1,2,3,5,8,13,\ldots$$

という数列になる。この数列は **Fibonacci 数列**として知られており、続く 2 つの値の和が次の値になっている性質を持っている。この数列の (先頭を 0 番目としたときの)n 番目の値を $\mathrm{fib}(n)$ と書くことにし、$\mathrm{fib}(n)$ を求める問題についてのアルゴリズムを考えよう。

5.1.1　再帰的アルゴリズム

Fibonacci 数列の k 番目の値と、それより前の値との関係を書くと次のようになる。

$$\mathrm{fib}(k) = \begin{cases} 1 & (k = 0 \text{ または } k = 1) \\ \mathrm{fib}(k-1) + \mathrm{fib}(k-2) & (k \geq 2) \end{cases} \tag{5.1}$$

この定義に従ってそのまま計算を行うのが最初のアルゴリズムである。関係が再帰的なので**再帰的アルゴリズム**と呼ぶことにしよう。このアルゴリズムに従った計算は、例えば $\mathrm{fib}(4)$ であれば次のような順序で進む。

$$\mathrm{fib}(4) \Rightarrow \mathrm{fib}(3) + \mathrm{fib}(2) \tag{5.2}$$

のように展開できるので、$\mathrm{fib}(3), \mathrm{fib}(2)$ をそれぞれ求める (☆) ことになる。$\mathrm{fib}(3)$ を展開するとと

$$\mathrm{fib}(3) \Rightarrow \mathrm{fib}(2) + \mathrm{fib}(1) \tag{5.3}$$

となるので、今度は $\mathrm{fib}(2), \mathrm{fib}(1)$ をそれぞれ求める (★) ことになる。$\mathrm{fib}(2)$ を展開してみると

$$\mathrm{fib}(2) \Rightarrow \mathrm{fib}(1) + \mathrm{fib}(0) \Rightarrow 1 + 1 \Rightarrow 2 \tag{5.4}$$

となる。また★で残されていた fib(1) は

$$\mathrm{fib}(1) \Rightarrow 1$$

である。そこで式 5.3 に戻って

$$\mathrm{fib}(3) \Rightarrow \mathrm{fib}(2) + \mathrm{fib}(1) \Rightarrow 2 + 1 \Rightarrow 3 \tag{5.5}$$

を得る。

次に☆で残っていた fib(2) を求めることになるが、ここは式 5.4 と全く同じ展開を行って

$$\mathrm{fib}(2) \Rightarrow \mathrm{fib}(1) + \mathrm{fib}(0) \Rightarrow 1 + 1 \Rightarrow 2 \tag{5.6}$$

となる。

ここでようやく式 5.2 に戻って、式 5.5 と式 5.6 の結果を使って

$$\mathrm{fib}(4) \Rightarrow \mathrm{fib}(3) + \mathrm{fib}(2) \Rightarrow 3 + 2 \Rightarrow 5$$

となる。

この再帰的アルゴリズムは、ほとんどそのまま Ruby の関数として定義することができる。アルゴリズムごとに関数名を変えて区別するために、`fibr(k)` という名前で定義すると、次のようになる。

```
1 def fibr(k)
2     if k==0 || k==1
3         1
4     else
5         fibr(k-1) + fibr(k-2)
6     end
7 end
```

ファイル 5.1: `fib.rb` (その 1・再帰的アルゴリズム)

練習 5.1 (再帰的アルゴリズムの使用) `fibr` を使って $\mathrm{fib}(k) > 100000$ となる最小の k を見つけよ。(Control C を押すと計算を中止できる。)

5.1.2　数え上げアルゴリズム

最初に説明したように、Fibonacci 数列は続く 2 つの値の和が次の値になっている。従って $\mathrm{fib}(0), \mathrm{fib}(1)$ の値から $\mathrm{fib}(2)$ を、$\mathrm{fib}(1), \mathrm{fib}(2)$ の値から $\mathrm{fib}(3)$ を求め…といった順に、直前の 2 つの値を覚えておけば下から順に数列を作ってゆくこともできる。

いま k 番目の Fibonacci 数を f, その 1 つ前の値を p1, もう 1 つ前を p2 とすると、この 3 つの値の関係は次の表のようにまとめられる。

k	0	1	2	3	4	5	6	7	8	9	10	11	12
fib(k): f	1	1	2	3	5	8	13	21	34	55	89	144	233
1 つ前: p1	–	1	1	2	3	5	8	13	21	34	55	89	144
2 つ前: p2	–	–	1	1	2	3	5	8	13	21	34	55	89

従って k 番目の Fibonacci 数を求める場合には、最初 f, p1 を 1, 1 としておき、

1. 次の p2, p1 の値をそれぞれ今の p1, f の値として、
2. 次の f の値を 次のp1 + p2 にする

という操作を $k-1$ 回繰り返せばよい。($k-1$ 回になるのは最初の f, p1 の値を $k=1$ のときのものにしたためである。)

このアルゴリズムを Ruby の関数として定義するには、f, p1, p2 という変数の値を適切に与えておき、上のような操作を繰り返すだけでよいので、次のようになる。

```
8  def fibl(k)
9      f=1
10     p1=1
11     for i in 2..k
12         p2 = p1       #fib(i-2)
13         p1 = f        #fib(i-1)
14         f  = p1 + p2  #fib(i)
15     end
16     f                 #fib(k)
17 end
```

ファイル 5.2: fib.rb (その 2・数え上げアルゴリズム)

注意すべき点としては、p2 = p1(12 行目) と p1 = f(13 行目) の順序である。p1= f を実行してしまうと p1 は「次の p1」の値になってしまう。一方、p2 = p1 を実行するときの p1 は「今の p1」の値が入っていなければいけない。従って 12 行目と 13 行目はこの順序で実行しなければいけない。

繰り返しが終わった後の f には fib(k) の値が入っているので、それを関数の答としているのが 16 行目である。

練習 5.2 (正しさの確認) いくつかの k について、fibr(k) と fibl(k) が同じ値を返すことを確認せよ。また、繰り返しを使ってある範囲の k の値すべてについて同じ値になっていることを確認するような関数を考えてみよ。

練習 5.3 (近似値との比較) 黄金比 $\frac{1+\sqrt{5}}{2} = \phi$ としたとき、fib(k) は

$$\mathrm{fib}(k) \simeq \frac{\phi^{k+1}}{\sqrt{5}} \tag{5.7}$$

と近似できることが知られている。式 5.7 に従って fib(k) の近似値を計算する関数 fiba(k) を定義し、いくつかの k について fibr(k) との誤差を調べよ。

練習 5.4 (代入の順序) ファイル 5.2 の、12 行目と 13 行目を入れ替えた場合、どのような値が計算されるかその理由とともに説明せよ。

5.2　計算時間の違いと計算量

5.2.1　スピード競争

さて、Fibonacci 数を計算するアルゴリズムを 2 つ紹介したので、どちらが速いかを比べてみよう。腕時計を使って時間を計ってもよいのだが、測定結果をグラフにする配布プログラム bench.rb を使うことにする。次の画面のように bench.rb を読み込み、そこに定義されている run を使って Fibonacci 数の計算をさせてみよ。

> 注意:　以下の画面を実行する前に、配布プログラム bench.rb をダウンロードしておく必要がある。

```
1  irb(main):004:0> load("./fib.rb")  # fibrなどの定義
2  => true
3  irb(main):005:0> load("./bench.rb")# runの定義
4  => true
5  irb(main):006:0> run("fibr", 10)   # fibr(10)を測る
6  fibr(10)...finished in 0.000000 seconds.
7  => 89
8  irb(main):007:0> run("fibl", 10)   # fibl(10)を測る
9  fibl(10)...finished in 0.000000 seconds.
10 => 89
```

5 行目の命令 run("fibr",10) は、式 fibr(10) を計算し、それにかかった時間を表示する。関数名を文字列として与える必要があることに注意して

ほしい。画面を見て分かるように、fibr, fibl ともに計算時間が非常に短く、速度の違いは分からない。

そこで $k = 10, 11, \ldots, 24$ と変化させて fibr(k), fibl(k) の計算時間がどう変わるかを見てみよう。次のように繰り返し命令 for を使えば自動的に実行させることができる。

```
irb(main):009:0> for k in 10..24
irb(main):010:1>   run("fibr", k)
irb(main):011:1>   run("fibl", k)
irb(main):012:1> end
fibr(10)...finished in 0.000000 seconds.
fibl(10)...finished in 0.000000 seconds.
fibr(11)...finished in 0.000000 seconds.
中略
fibl(23)...finished in 0.000000 seconds.
fibr(24)...finished in 0.420000 seconds.
fibl(24)...finished in 0.000000 seconds.
=> 10..24
```

もし $k = 24$ のときでも fibr の計算時間がほとんど 0 であった場合は、11 行目の 24 をより大きな値に変えて実行してみよ。

$k = 24$ くらいになると計算時間の差が見えてきた。関数 run は時間を測ると同時に、測った時間を図 5.1 のようなグラフとして表示してくれる。グラフは k の値を横軸に、時間を縦軸にとったもので、fibr の計算時間は k が大きくなるに従って除々に大きくなってきていることが分かる。一方、fibl はほとんどゼロのままである。つまり、

k が大きくなるにつれ、fibr(k) の計算時間は fibl(k) よりも長くなる

ことが分かる。

5.2.2　実行時間から計算時間を見積る

次に、例えば fibr(100) を計算するのにどのくらい時間がかかるかを予想してみよう。以下では fibr(k) の計算時間を $t_r(k)$ と書くことにする。

図 5.2 は、k の範囲を広げて fibr(k) の計算時間を測ってグラフにしたものである。k が大きくなると時間が急激に増えてゆくので時間軸を対数目盛りにしてみると、計算時間はほぼ直線的に増えていることが分かる。時間が対数目盛りなので、

方法は第 5.5 節 (p. 84) の「グラフの表示方法の変更」を参照せよ。

fibr(k) の実行時間は k の指数関数に近似できる

つまり A, B を定数として $t_r(k) = A \cdot B^k$ と近似できると推測できる。

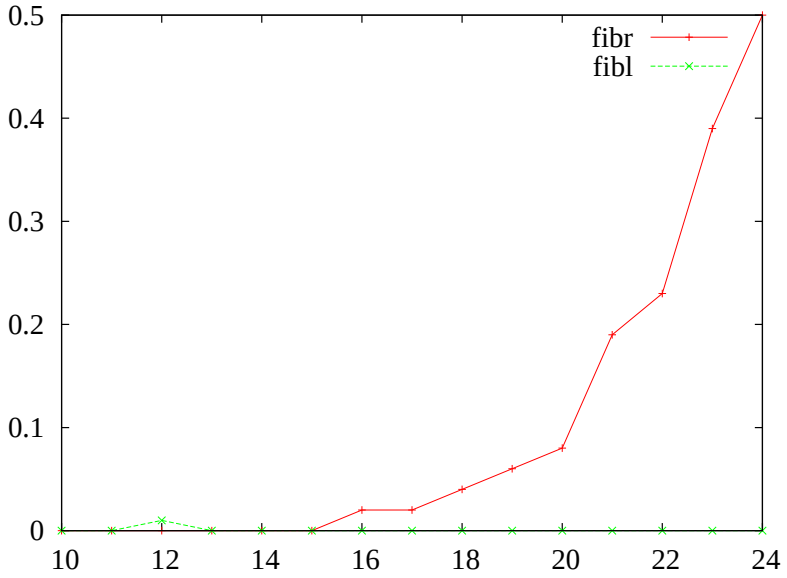


図 5.1: $10 \leq k \leq 24$ に対する `fibr(`k`)` と `fibl(`k`)` の実行時間

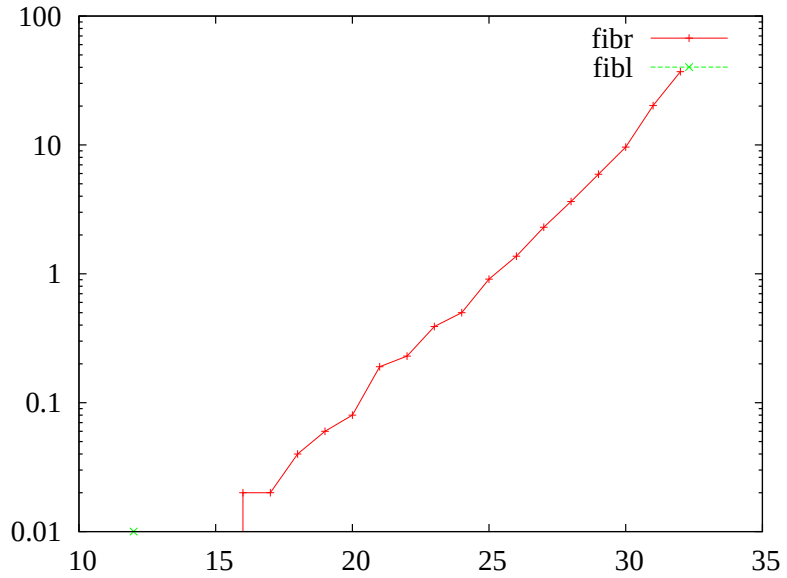


図 5.2: $10 \leq k \leq 32$ に対する `fibr(`k`)` の計算時間 (時間軸は対数目盛りになっている)

この式と実際の計算時間から A, B を求め、それを使って `fibr(100)` の計算時間を予想してみよう。いま、実行時間が $t_r(25) = 0.86$ 秒、$t_r(32) = 35.26$ 秒だったとする。これを上の式にあてはめると $A = 1.5 \times 10^{-6}$,　$B = 1.7$ となる。つまり、

$$t_r(k) \simeq 1.5 \times 10^{-6} \times 1.7^k$$

という近似式を得る。従って `fibr(100)` の予想計算時間は $1.5 \times 10^{-6} \times 1.7^{100} \simeq 1.7 \times 10^{17}$(秒) $\simeq$ 53 億 (年) となる。

次に `fibl(`k`)` の計算時間について考えてみよう。これまでの測定では `fibl` の計算時間はほとんど 0 だったので、より大きな k ついて時間を測定してみることにする。ここで `fibl` のかわりに **Fibonacci 数の下位 6 桁だけを求める** `fibl6` を定義し、それの計算時間を測ることにする。

桁数の多い数の足し算は、その桁数に比例する時間がかかる。k が大きくなると Fibonacci 数は非常に大きな数になるため、それを求めるための足し算にかかる時間の変化も無視できなくなる。6 桁程度の数の足し算であれば、それにかかる時間は一定になる。

まず `fibl6` の定義であるが、この関数は `fibl` の定義 (ファイル 5.2) の 14 行目の

```
f = p1 + p2
```

を

```
f = (p1 + p2) % 1000000
```

に変更するだけでよい。

`fibl6` が定義できたら、計算時間を測ってみよう。以下は k を 10 万、20 万、...100 万と変化させたときの画面である。

```
23 irb(main):026:0> reset()
24 => []
25 irb(main):027:0> for m in 1..10
26 irb(main):028:1>   run("fibl6", 100000*m)
27 irb(main):029:1> end
28 fibl6(100000)...finished in 0.390000 seconds.
29 fibl6(200000)...finished in 0.770000 seconds.
30 中略
31 fibl6(900000)...finished in 3.440000 seconds.
32 fibl6(1000000)...finished in 3.820000 seconds.
33 => 1..10
```

23 行目の `reset()` は、いままで描いたグラフのデータを消去して、新しいグラフを作成させるための命令である。実行結果のグラフ (図 5.3) を見ると、計算時間 $t_l(k)$ はほぼ直線になっている。(縦軸は通常の目盛りに戻してあることに注意せよ。) これより、

`fibl6(`k`)` の実行時間は k の一次関数に近似できる

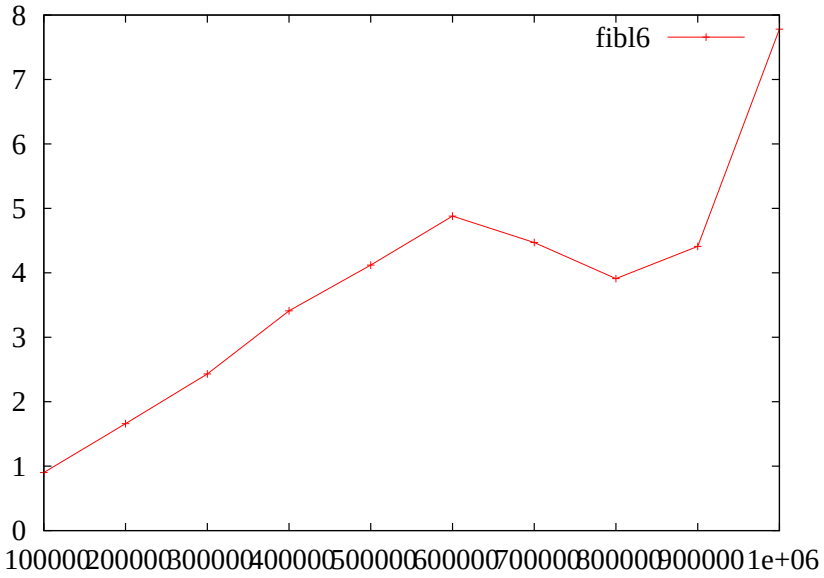


図 5.3: 10 万 $\leq k \leq$ 100 万 に対する `fibl6(`k`)` の計算時間

つまり C を定数として $t_l(k) = ck$ に近似できると推測できる。

今回の測定値では $C \simeq 7.8$ つまり $t_l(k) \simeq 7.8 \times 10^{-6} k$ となる。

練習 5.5 (計算時間の実測) 関数 `fibr(`k`)`, `fibl6(`k`)` の計算時間を実測し、$t_r(k) \simeq A \cdot B^k$, $t_l(k) \simeq Ck$ にあてはまる A, B, C を求めよ。

5.2.3　アルゴリズムから計算時間を見積る

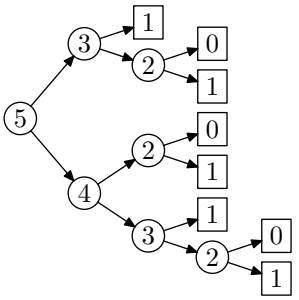
ここまでの話は、**プログラム**を実際に動してみた結果から計算時間を予想していたに過ぎない。つまり計算時間を予想したくても、実際にプログラムを作りそれを動かさなければいけない。これではプログラムを作る前に良いアルゴリズムを選ぶことはできない。

そこで今度は、**アルゴリズム**だけから計算時間を見積ってみよう。つまりアルゴリズムの性質から演算や式展開などの回数を理論的に求め、そこから計算時間を見積るという方法である。これであれば、プログラムを作る前からアルゴリズムによる計算時間の違いを予想できることになる。

再帰的アルゴリズムの計算時間

再帰的な Fibonacci 数の計算アルゴリズムは、(a) k が 1 より大きいかどうかを判定すること、(b) $\mathrm{fib}(k)$ という式を $\mathrm{fib}(k-1) + \mathrm{fib}(k-2)$ へ展開することと、(c) $\mathrm{fib}(k-1)$ と $\mathrm{fib}(k-2)$ の値を足すことから成り立っている。

いま fib(5) の計算を下図のように表わしてみる。⑤ は fib(5) の計算を表わし、○はさらに展開された計算、□はそれ以上展開されなかった計算を示す。



アルゴリズムとの対応を考えると、(a) は○と□の合計の個数回、(b) と (c) はそれぞれ○の個数回だけ行われている。また、どんな k であっても□の個数は $\mathrm{fib}(k)$ の値になり、○の個数は $\mathrm{fib}(k)-1$ の値になる。計算時間 $t_r(k)$ は、(a),(b),(c) の 1 回あたりの時間をそれぞれ T_a, T_b, T_c とすれば

$$t_r(k) = T_a(2\mathrm{fib}(k)-1) + (T_b+T_c)(\mathrm{fib}(k)-1) \tag{5.8}$$

となる。ところで、練習 5.3 で見た近似式を使うと式 5.8 はさらに

$$t_r(k) \simeq \frac{(2T_a+T_b+T_c)\phi}{\sqrt{5}}\phi^k - (T_a+T_b+T_c) \tag{5.9}$$

と近似できる。

T_a, T_b, T_c は、実行するコンピュータの性能や、プログラミング言語処理系によって異なるが、一定の値をとるとすればこの式は

再帰的アルゴリズムによる Fibonacci 数 $\mathrm{fib}(k)$ の計算時間は ϕ^k に比例した式で近似される

ことを示している。

第 5.2.2 節で実際の計算時間から求めた近似式は $t_r(k) \simeq 1.5\times10^{-6}\times1.7^k$ であったが、この式の k の底と式 5.9 の k の底 $\phi \simeq 1.618$ が近い値であることからもこの見積りの妥当性が分かる。

数え上げアルゴリズムの計算時間

数え上げアルゴリズムの場合の計算時間はもっと簡単である。このアルゴリズムでは、(d) 最初に「現在、1 つ前、2 つ前の値」を決めた後は、(e) 繰り返しが終わったかどうかの判定 (f) 「現在、1 つ前、2 つ前の値」の入れ換え (g) 現在の値の計算 (足し算) を $k-1$ 回繰り返しているだけである。従って計算時間 $t_l(k)$ は (d), (e), (f), (g) の 1 回あたりの時間をそれぞれ T_d, T_e, T_f, T_g としたときに

$$t_l(k) = T_d + (T_e+T_f+T_g)(k-1) \tag{5.10}$$

である。この式は

数え上げアルゴリズムによる Fibonacci 数 $\mathrm{fib}(k)$ の計算時間は k に比例した式で近似できる

ことを示している。実際、第 5.2.2 節で測定したプログラム `fibl6` の計算時間はほぼ k に比例していたので、この見積りに合致していると言える。

5.2.4　計算量

アルゴリズムの速さを比べる場合には、式 5.9 や式 5.10 の T_a, T_b などどをすべて 1 にした、**計算量**と呼ばれる値を用いる。Fibonacci 数を求める 2 つのアルゴリズムであれば、それぞれの計算量 $c_r(k), c_l(k)$ は次のようになる。

$$\begin{aligned} c_r(k) &\simeq \frac{3\phi}{\sqrt{5}}\phi^k - 3 \qquad \text{(再帰的)} \\ c_l(k) &= 5k + 2 \qquad \text{(数え上げ)} \end{aligned}$$

多くの場合は、さらにこれらの式の

- 定数係数や定数項を消し、
- 入力の大きさを表わす変数について、最も次数が高い項のみを残した

ような式を求めた上で比較をする。このような式は**計算量のオーダー**と呼ばれ、入力を大きくした際に、計算量がこの式に近似できるような変化をすることを意味する。Fibonacci 数の再帰的アルゴリズムであれば、ϕ^k の係数 $\frac{3\phi}{\sqrt{5}}$ や定数項 -3 を消し、数え上げアルゴリズムであれば係数 5 や定数項 2 を消して

$$\begin{aligned} &c_r(k) \text{ のオーダーは } \phi^k \\ &c_l(k) \text{ のオーダーは } k \end{aligned}$$

となる。

また、アルゴリズムによっては「ある場合は k^2 回、別の場合は k 回」のように場合によって計算量が違うこともある。そのような場合には最悪の場合として**最も大きくなる場合だけを考える**(この例では k^2) のが一般的である。

このように最悪の場合の計算回数を考えて、その最も変化の大きな項を取り出したことを示すためにO(オー)記法が用いられる。Fibonacci 数のアルゴリズムであれば次のように書くことができる。

$$\begin{aligned} c_r(k) &= O(\phi^k) \\ c_l(k) &= O(k) \end{aligned}$$

さて、計算量のオーダーは実際の計算時間の近似としてはかなり粗いものであるが、これを使ってアルゴリズムを比べても大丈夫なのだろうか。例え

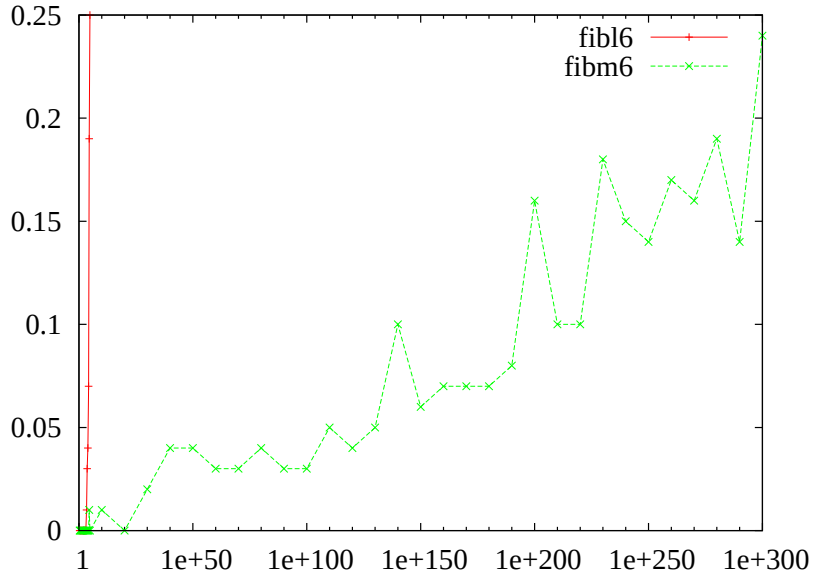


図 5.4: $k \leq 10^{300}$ に対する `fibm6`(k) の計算時間

ば $O(\phi^k)$ と $O(k)$ であれば、$O(\phi^k)$ の方が変化が大きいので悪いアルゴリズムだと結論するわけだが、実際の計算時間は $O(k)$ の方が 100 倍多く操作をしているかも知れず、実際には $O(\phi^k)$ のものの方が速いことがあるかも知れない。

しかし、計算量のオーダーの違いはそれ以上に大きい。下の表は k と ϕ^k の変化を示したものであるが、k が大きくなると ϕ^k は圧倒的に大きな数になる。これならば定数倍の違いを無視しても構わないことが分かるだろう。

k	1	10	100	1000	10000
ϕ^k	1.6	1.2×10^2	7.9×10^{20}	9.7×10^{208}	7.5×10^{2089}

5.3　行列を用いた Fibonacci 数のアルゴリズム

Fibonacci 数を求めるアルゴリズムとして、計算量が $O(\log k)$ のものを紹介しよう。つまり $\mathrm{fib}(k)$ を求めるのに k の桁数に比例した時間しかかからないものである。

これから紹介するアルゴリズムに従って作られた関数 `fibm6` の計算時間を図 5.4 に示す。このグラフは横軸が対数目盛りになっている。グラフがほぼ直線になっていることから、計算時間が $\log k$ に比例していることが分かる。一緒に `fibl6` の計算時間も描かれているが、`fibm6` の方が大幅に速いことも分かるだろう。

5.3.1　行列積による求解

このアルゴリズムでは連続する 2 つの Fibonacci 数をベクトル $v_k = \begin{pmatrix} \mathrm{fib}(k+1) \\ \mathrm{fib}(k) \end{pmatrix}$ として考える。すると $\mathrm{fib}(k)$ の定義から v_k と v_{k+1} の間の関係は行列 $Q = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ を使って $v_{k+1} = Qv_k$ と表わせる。これより v_n は v_0 を Q に n 回掛けたもの、つまり $v_n = Q^n v_0$ と導かれる。

練習 5.6 (行列積による求解の正しさ) $v_{k+1} = Qv_k$ を証明せよ。

5.3.2　冪乗の計算アルゴリズム

ここで行列 Q の冪乗の計算方法について考える。単純には行列の掛け算を n 回行えば Q^n が求まる。しかしこれでは計算量は $O(n)$ になってしまい、数え上げアルゴリズムと変わらない。

実は、もっと計算量の小さな冪乗計算アルゴリズムがある。例として Q^{16} で説明する。$Q^{nm} = (Q^n)^m$ であるので $Q^{16} = (((Q^2)^2)^2)^2$ と変形できる。自乗の計算は 1 回の掛け算でできるので、Q^{16} が 4 回の行列の掛け算でできるということである。単純に計算すれば 16 回の行列の掛け算が必要だったのでかなり回数を減らせている。

この考え方を一般化すると次のような関係として定義できる。

$$Q^n = \begin{cases} E & (n = 0) \\ (Q^{n/2})^2 & (n \text{ は 2 以上の偶数}) \\ Q \times Q^{n-1} & (n \text{ は奇数}) \end{cases} \tag{5.11}$$

(ただし E は単位行列。) つまり、Q^n は n が偶数のときは $Q^{n/2}$ を求めてからそれを自乗し、奇数のときは Q^{n-1} に Q を掛けるという意味である。

この関係を使えばどのような n に対しても少ない回数の掛け算で Q^n を求める式が導ける。例えば Q^{20} であれば

$$Q^{20} \Rightarrow (Q^{10})^2 \Rightarrow ((Q^5)^2)^2 \Rightarrow ((Q \times Q^4)^2)^2 \Rightarrow ((Q \times (Q^2)^2)^2)^2 \Rightarrow$$
$$((Q \times ((Q^1)^2)^2)^2)^2 \Rightarrow ((Q \times ((Q \times Q^0)^2)^2)^2)^2 \Rightarrow ((Q \times ((Q \times E)^2)^2)^2)^2$$

のように展開できる。掛け算の回数は $\times$ と自乗の個数なので 6 回の行列の掛け算で求められている。

練習 5.7 (行列の冪乗) 行列 A の冪乗 A^n を計算する関数 `matpower(a,n)` を次のようにして定義せよ。行列は 2×2 のものだけを扱うことにして、2 次元配列で表わすことにする。

a) 行列 a, b の積を求める matmul(a,b) を定義せよ。ヒント: 行列を 2×2 に限っているので次のような計算をするだけである。

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}\begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00}b_{00}+a_{01}b_{10} & a_{00}b_{01}+a_{01}b_{11} \\ a_{10}b_{00}+a_{11}b_{10} & a_{10}b_{01}+a_{11}b_{11} \end{pmatrix}$$

「行列 (matrix) を掛ける (multiply)」ことが分かるように matmul という関数名にしている。matsquare, matpower も同様の方法で名前がついている。

b) 行列 a の自乗を求める matsquare(a) を matmul を用いて定義せよ。

c) 式 5.11 に従って行列 a の n 乗を求める matpower(a,n) を定義せよ。

練習 5.8 (行列を用いた Fibonacci 数の計算) 練習 5.7 で定義した matpower(a,n) を用いて $\mathrm{fib}(k)$ を求める関数 fibm(k) を定義せよ。また、fibm(k) と fibl(k) が同じ答を出すことをいくつかの k について確認せよ。

5.3.3　行列を用いたアルゴリズムの計算量

最後に、この行列を用いたアルゴリズムの計算量を考えよう。今回の計算は Q^k の計算に帰着されるので、冪乗の計算アルゴリズムの計算量を考えればよい。

Q^k が式 5.11 に従って 1 回展開されると、行列の掛け算は 1 回増え、k は (奇数のとき)1 減るか (偶数のとき) 半分になるかのどちらかである。従って k が 0 になるまで何回展開されるか、つまり k がどのように減ってゆくかを考えればよい。k の減り方は一定ではないが、1 番遅い減り方をするのは 奇数 $\Rightarrow$ 偶数 $\Rightarrow$ 奇数 $\Rightarrow$ 偶数 $\Rightarrow \cdots$ のように減ってゆく場合なので、$k = 2^m - 1$ という値になっている場合である。このときに $2m$ 回つまり $2\log_2(k+1)$ 回の展開が起きるのが最悪の回数である。2×2 行列の掛け算 1 回に必要な基本的計算は (桁数が大きくないのであれば) 定数回なので、オーダーを考える場合には正確な回数は考えなくともよい。結局、このアルゴリズムの計算量 $c_m(k)$ は

$$c_m(k) = O(\log k)$$

であることが分かる。

練習 5.9 (行列を用いた Fibonacci 数の計算 (下 6 桁)) 練習 5.8 で定義した関数を変更し、Fibonacci 数の下位 6 桁のみを求める fibm6 を定義せよ。時間計測用の配布プログラムを用いて、大きな k に対する fibm6(k) の計算時間が $\log k$ に比例していることを確認せよ。

5.4　整列のアルゴリズム

沢山のデータを順番に並べ替えることを**整列**(sorting) という。コンピュータを使った情報処理では、大量のデータの整列が頻繁に行われるので、良いアルゴリズムは大量データ処理に不可欠である。

ここでは整列のアルゴリズムの中から簡単なものとして、**単純整列法**と**併合整列法**を紹介する。

以下では、数値がしまわれている大きさ n の配列があったときに、その数値を小さい順に並べ替える問題を考えることにする。

5.4.1　単純整列法

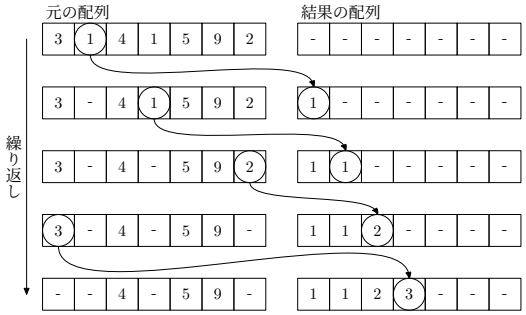


図 5.5: 単純整列法の考え方

単純整列法は図 5.5 のように「元の配列の中で 1 番小さな数を見つけてそれを取り除き、結果の配列の最後に置く。」という作業を元の配列が空になるまで繰り返す方法である。

ただし、図 5.5 のような方法をそのまま使うと、配列を 2 つ用意しなければいけない上に、元の配列では「取り除かれた値」を記録しておく必要がある。

この方法を少し工夫すると、元の配列だけを使って整列ができる。これは、図 5.6 のように「元の配列の中で 1 番小さな数を見つけそれを配列の 0 番目と入れ替える。次に配列の 1 番目以降で 1 番小さな数 (つまり全体では 2 番目に小さな数) を見つけ、配列の 1 番目に場所に置く」という作業を繰り返すものである。図中では、入れ替えが終わったものが灰色に塗られ、白色の中から 1 番小さな数が選ばれて入れ替えられている。

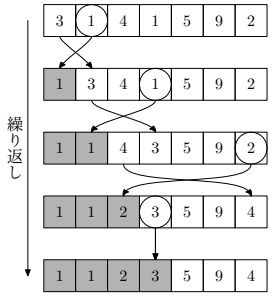


図 5.6: 工夫した単純整列法

この工夫した単純整列法をプログラムにするとファイル 5.3 のようになる。

```
1 def simplesort(a)
2   for i in 0..(a.length()-1)
3     k = min_index(a,i)
4     v = a[i]
5     a[i] = a[k]
6     a[k] = v
7   end
8   a
9 end
```

ファイル 5.3: simplesort.rb (一部)

i 回目の繰り返しでは、0 番目から i − 1 番目までは整列されているので、i 番目以降で最小の値が現われる場所を探し (3 行目の min_index(a,i)) その場所を k とする。元々i 番目にあった値を v とする (4 行目)。k 番目の値、つまり最小の値を i 番目にしまい (5 行目)、元々i 番目にあった値 v を k 番目へどける (6 行目) と入れ替えることができる。

練習 5.10 (単純整列法) 配列 a の (先頭を 0 番目としたときの)i 番目以降の値の中で、最小値が出現する番号を答える min_index(a,i) をファイル 5.3 に追加して、単純整列法を完成させよ。

```
1 irb(main):004:0> a=[3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3]
2 => [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3]
3 irb(main):005:0> min_index(a,0)
4 => 1
5 irb(main):006:0> min_index(a,4)
6 => 6
7 irb(main):007:0> simplesort(a)
8 => [1, 1, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 9, 9, 9]
```

単純整列法の計算量

配列の大きさを n として単純整列法の計算量を考えてみよう。このアルゴリズムは全体で n 回の繰り返しをしている。i 番目の繰り返しでは $n-i$ 個の値から最小値を見つける必要があるので、$n-i$ 回の繰り返しが必要になる。つまり、全体では

$$\sum_{i=0}^{n-1}(n-i) \quad = \quad n+(n-1)+\cdots+1 \quad = \quad \frac{1}{2}n(n-1)$$

回の繰り返しということになる。従って**単純整列法の計算量は $O(n^2)$ である**。

5.4.2　併合整列法

単純整列法の計算量は $O(n^2)$ だった。これは、1000 万個のデータを並べ替えるには 1000 万の自乗、つまり 100 兆回程度の繰り返しをしなければいけないということである。コンピュータが 1 秒間に 10 億回の繰り返しを行えるとしても、10 万秒も時間がかかってしまう。

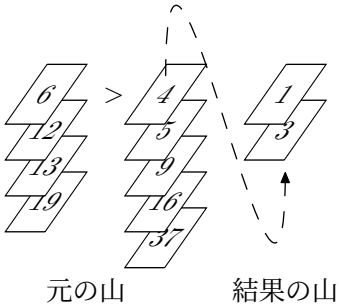


図 5.7: 2 つのカードの山の併合

もっと良い計算量の整列法の 1 つとして**併合整列法**を紹介しよう。この方法の基本は図 5.7 のように、小さい順に並んでいる 2 つのカードの山を混ぜ合わせて小さい順に並んだ 1 つの山にする、**併合**(merge) という作業である。元の 2 つの山は小さい順に並んでいるので、この作業は 1 番上にあるカードの値を比べて、小さい方のカードを結果の山の 1 番下に移すことを繰り返してゆき、一方の山のカードがなくなったらもう一方の山の残りのカードをそのまま結果の山の 1 番下につなげる。これで小さい順に並んだ結果の山を得ることができる。

予め小さい順に並んだ山が 2 つあれば併合によって整列された山を作れることは分かった。この「小さい順に並んだ山」はどうやって作るのかというと、やはり併合によって作ればよい。つまり、図 5.8 の左端の列のように、バラバラに並んでいるカードそれぞれをカード 1 枚だけの山だと思い、2 つずつを併合して 2 枚のカードの山を作る。できた山を再び 2 つずつ併合して 4 枚のカードの山を作る。この手順を繰り返してゆけば図の右端のように 1 つの小さい順に並んだ山になるというわけである。

併合整列法の計算量

プログラムを作るより前に、併合整列法の計算量を考えておこう。この方法は、併合を何度も行っているので、一見無駄なことをしているようにも思えるが、果たして単純整列法よりも速いのだろうか。

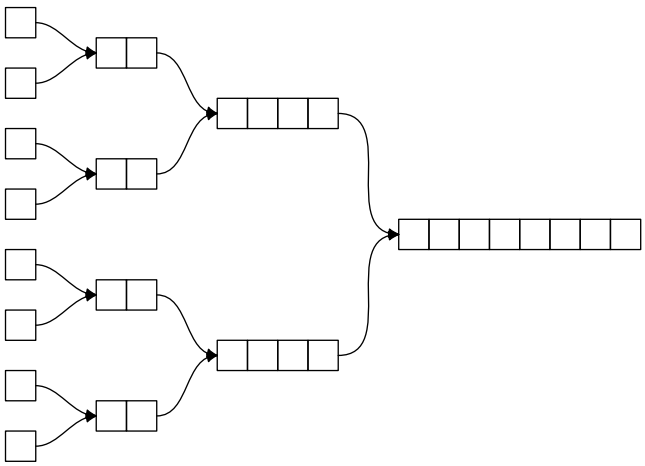


図 5.8: 併合の繰り返し

まず併合の計算量を考える。ℓ 枚のカードの山を 2 つ併合するには、2 つの山の 1 番上のカードを比較して、どちらかのカードを移動させることを 2ℓ 回繰り返すだけである。

次に、1 枚のカードの山が n 個ある状態から、併合を繰り返して n 枚のカードの山 1 個にするまでの手順を考えよう。以下、簡単のために $n = 2^e$ だとする。すると、(最初を 1 回目だとして)i 回目の繰り返しは2^{i-1}枚の山を2つ併合して2^i枚の山を得ることを 2^{e-i} 個の組について行うことになる。下線の作業は 2^i 回の繰り返しであるので、結局、i 回目の繰り返しは $2^i \times 2^{e-i} = 2^e = n$ に比例した作業を行うことになる。この繰り返しは i が 0 から $e-1$ までの e 回行われるので、全体では $n \times e = n \log_2 n$ 回の繰り返しとなる。つまり、**併合整列法の計算量は** $O(n \log n)$ であり、単純整列法の $O(n^2)$ よりも良いことが分かる。

例えば 1000 万個のデータを並べ替えを考えると、1000 万 $\times \log_2$(1000 万) $\simeq$ 2 億 3000 万 回程度の繰り返しになる。単純整列法は $O(n^2)$ だったので 100 兆回程度の繰り返しをしなければいけなかったので、劇的に速くなっていることが予想できる。

併合整列法のプログラム

さてそれでは併合整列法を Ruby の関数として定義してみよう。このアルゴリズムでは「カードの山」を使っていたが、これは Ruby の配列として表わすことにする。

2 つの配列 a, b を併合して 1 つの山にする関数 `merge(a,b)` はファイル 5.4 のように定義できる。

この関数では、まず a と b の合計の長さを持つ配列 c を作る (2 行目)。次に、

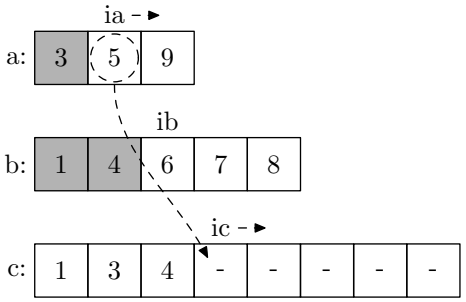
```
1  def merge(a,b)
2    c = Array.new(a.length()+b.length())
3    ia=0
4    ib=0
5    ic=0
6    while ia < a.length() && ib < b.length()
7      if a[ia] < b[ib]
8        c[ic] = a[ia]
9        ia = ia + 1
10       ic = ic + 1
11     else
12       c[ic] = b[ib]
13       ib = ib + 1
14       ic = ic + 1
15     end
16   end
17   a,b のうちまだ残っている方を全て c に移す (省略)
18   c
19 end
```

ファイル 5.4: mergesort.rb (前半)

- 配列 a から移動した値の個数を覚える変数 ia、
- 配列 b から移動した値の個数を覚える変数 ib、
- 配列 c に移動された値の個数を覚える変数 ic

を用意する。

あとは「配列 a, b のどちらか一方から配列 c に値を一つ移動させる」作業の繰り返しである (6 行目から 16 行目)。繰り返し 1 回の作業は、次の図のように配列 a と b の (先頭を 0 番目として)ia, ib 番目の値を比べ (7 行目)、小さい方を c の ic 番目に移し、ia, ib の移動した方と ic をそれぞれ 1 増やす (8 行目から 14 行目)。



この繰り返しは、a と b のどちらか一方の値をすべて移し終えるまで (つまり 6 行目の条件のように a と b の両方にまだ移していない値が残っている限り) 続ける。

16 行目までの繰り返しが終わったときは、配列 a または b のどちらか一方の値はすべて移し終え、もう一方にはまだ値が移し終わっていない状態になる。(上の図であれば配列 b の値をすべて移し終わり、a には最後の 9 が残る。) そこで、値が残っている方の配列の残りの値をすべて c に移す繰り返しを行う。ファイル 5.4 では 17 行目の部分だが、省略してある。

最後に配列 c を答として返す (18 行目)。

練習 5.11 (併合) ファイル 5.4 の省略された部分を補って関数 merge を完成させよ。

```
1 irb(main):004:0> merge([3,5,9],[1,4,6,7,8])
2 => [1, 3, 4, 5, 6, 7, 8, 9]
3 irb(main):005:0> merge([0,0.5,1.0],[0,0.9,1.0])
4 => [0, 0, 0.5, 0.9, 1.0, 1.0]
```

バラバラの順序の配列から、併合を繰り返して整列する関数 mergesort(a) はファイル 5.5 のようになる。

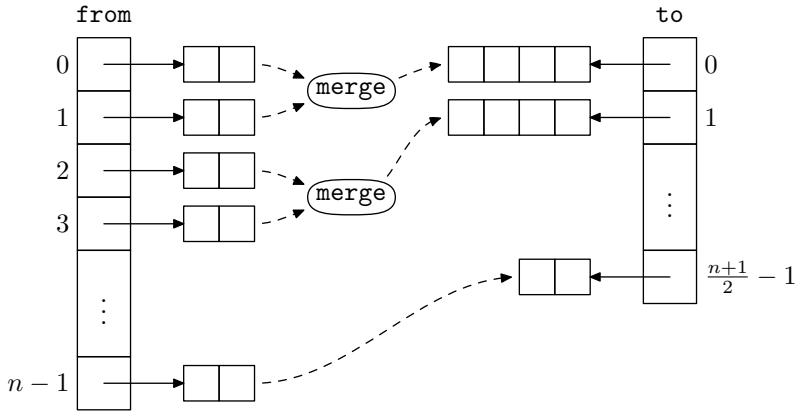
この関数の前半 (25 行目まで) は「カード 1 枚ずつの山」を作っている部分である。与えられた配列 a の大きさを n として、n 行 1 列の配列 from を作り、from[i][0] に a[i] の値をしまっている。24 行目 の式 [(a[i])] は、大きさ 1 の配列を作ることに注意してほしい。

後半は、from にしまわれている n 個の配列を 2 つずつ merge によって併合し、配列の個数を半分に減らしてゆく作業を繰り返している。

繰り返しの各回の作業を次の図に従って説明しよう。

```
20 def mergesort(a)
21   n = a.length()
22   from = Array.new(n)
23   for i in 0..(n-1)
24     from[i] = [ (a[i]) ]
25   end
26   while n > 1
27     to = Array.new((n+1)/2)
28     for i in 0..(n/2-1)
29       to[i] = merge(from[i*2], from[i*2+1])
30     end
31     if !is_even(n)
32       to[(n+1)/2-1]=from[n-1]
33     end
34     from=to
35     n=(n+1)/2
36   end
37   from[0]
38 end
```

ファイル 5.5: mergesort.rb (後半)



- まず、from の半分の大きさの配列 to を作る。式 (n+1)/2 は n が奇数のとき、n を 2 で割って切り上げた値になる。(図は n が奇数の場合である。)
- 次に from の 2i 番目と 2i + 1 番目を merge によって併合し、to の (先頭を 0 とした)i 番目にしまう (29 行目)。この作業は i が n/2-1 になるまで繰り返すことになっている (28 行目) が、n が奇数のときは余りが

切り捨てられるので、from[n-2] までが併合されて、from[n-1] は処理されないことになる。

- n が奇数のときは、from の最後の配列を to の最後にしまう (32 行目)。
- このようにして作られた配列の配列 to を、次回の from にし (34 行目)、長さ n を半分 (の切り上げ) にする (35 行目)。

最終的に n が 1, つまり 1 個の配列に併合されたら from[0] にしまわれている配列が整列された結果となる (37 行目)。

練習 5.12 (併合整列法の実行) 練習 5.11 で完成させた merge とファイル 5.5 をあわせて、併合整列法による整列を行ってみよ。

```
1 irb(main):007:0> a=[3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3]
2 => [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3]
3 irb(main):008:0> mergesort(a)
4 => [1, 1, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 9, 9, 9]
```

練習 5.13 (整列法プログラムの速度の比較) 単純整列法と併合整列法のプログラムを見比べると、併合整列法では併合のための配列を作るといったより複雑な作業をしている。そのため、配列が短い場合は simplesort の方が、長くなると mergesort の方が速いことが予想される。以下のようにして実際の計算時間を比較せよ。

a) 整列させる配列を作るために、配布プログラム randoms.rb に含まれている randoms(id,size,max) を使う。

関数 randoms の詳しい説明は第 5.5 節にある。

b) 計算時間を測るために、第 5.2.1 節で紹介した配布プログラム bench.rb の関数 run(f,x,v) を使う。

この run の使い方は第 5.2.1 節のときと少し違う。詳しくはやはり第 5.5 節を参照せよ。

c) 2 つの整列法プログラム比較する関数 compare_sort(m,s) を次のように定義する。この関数は、大きさが $s, 2s, 3s, \ldots, m$ であるようなデタラメな配列を順に 2 つずつ作り、simplesort と mergesort を使って整列させて時間を測定している。

この compare_sort(max,step) を使って、計算時間の違いを実際に比較せよ。

run が測定する時間の精度は高くないので、正確な比較のためには、同じ計算を 1000 回繰り返すような関数を作り、その計算時間を計るといった工夫が必要になる。

5.5　定義のまとめ

計算時間の測定: (配布プログラム bench.rb) 関数 run は、関数の計算時間を測定し、それをグラフにする。

```
load("./randoms.rb")     # randoms(id,size,max)
load("./bench.rb")       # run(function_name, x, v)
load("./simplesort.rb") # simplesort(a)
load("./mergesort.rb")  # mergesort(a)

def compare_sort(max, step)
  for i in 1..(max/step)
    x=i*step
    a=randoms(i,x,1)
    run("simplesort", x, a)
    a=randoms(i,x,1)
    run("mergesort", x, a)
  end
end
```

ファイル 5.6: compare_sort

- run("fibr",n) のように実行した場合は、fibr(n) の計算を行い、その際の計算時間を X 座標が n の位置に表示する。
- run("fibr",x,n) のように実行した場合は、fibr(n) の計算を行い、その際の計算時間を X 座標が x の位置に表示する。

グラフの表示方法の変更: (配布プログラム bench.rb) command という命令を用いてグラフの表示方法を変更できる。主な命令には次のものがある。

- command("set logscale y") — グラフの Y 軸を対数スケールにする。y を x にすると X 軸が対数スケールになる。
- command("unset logscale") — 対数スケールをやめる
- command("set xrange [a:b]") — X 軸の範囲を a から b までにする。xrange を yrange にすると Y 軸の範囲を変更する。
- command("set autoscale") — 表示範囲を自動的に変更する。

実際には" "の内側は、グラフの表示に用いている GNUPLOT というソフトウェアに対する命令である。他の命令については GNUPLOT のマニュアルなどを参考にしてほしい。

また reset() という命令を実行すると、記録していたデータを全て消去して、以降の run から新しいグラフを作成する。

デタラメな中身の配列を作る: (配布プログラム randoms.rb) 関数 randoms(id,size,max) は、大きさ size の配列を作る。中身は 0 以上 max 未満のデタラメな値

であり、max が 1 のときは実数、1 より大きいときは整数になる。整数 id は、系列番号を意味し、id, size, max がすべて同じであれば同じ内容の配列が作られ、id が違えば異なる内容になる。

5.6　章末問題

練習 5.14 (組み合わせ数の計算量)　a) 組み合わせ数の計算を練習 4.2(p. 52) の定義に従って再帰的に行う場合の計算量を求めよ。

b) 同じ計算を第 4.3.1 節のように繰り返しによって行う場合の計算量を求めよ。

練習 5.15 (言葉探しの計算量) 文字列 s 中に文字列 p が現われる位置を探す方法として、第 4.3.2 節 (p. 55) で紹介したものがある。s と p の長さ n, m を入力の大きさだとして、これの計算量を n, m で表わせ。

ヒント: 最悪の場合を考えればよい。s が「$\overbrace{\texttt{aa}\cdots\texttt{a}}^{n-1}\texttt{b}$」、p が「$\overbrace{\texttt{aa}\cdots\texttt{a}}^{m-1}\texttt{b}$」という文字列だったときの submatch と match の繰り返し回数を考えよ。

練習 5.16 (同じ誕生日の人達) 同じ年に生まれた n の学生がいる。この中には、同じ誕生日を持つ学生と、そうでない学生がいる。前者が何人いるか数えるアルゴリズムをいくつか考え、その計算量を考えよ。簡単のために、誕生日は 1 月 1 日からの日数 (つまり 0 から 364 の整数) で表わすことにする。また、学生の誕生日は大きさ n の配列 b に入っているが、その順序はデタラメだとする。

アルゴリズム 1: 同じ誕生日を持つ学生の人数を変数 m で表わすことにして、最初は 0 にしておく。学生の誕生日を 1 人ずつとり出し、配列 b のすべての中身と比較して同じ誕生日の学生がいるかどうかを調べ、1 人でも同じ誕生日の者がいた場合は m を 1 増やす。これをすべての学生について行う。

アルゴリズム 2: 同じ誕生日を持つ学生の人数を変数 m で表わすことにして、最初は 0 にしておく。配列 b を併合整列法によって小さい順に並べる。並べ換えた配列を先頭から順に調べてゆき、同じ値が連続する区間の長さ k を数える。k が 2 以上の場合の区間があった場合は、m を k だけ増やす。これを配列の最後まで行う。

アルゴリズム 3: 日付ごとの人数を数えるために、大きさ 365 の配列 c を作り、中身をすべて 0 にしておく。学生の誕生日 d を 1 人ずつとり出し、その日付の人数 c[d] を 1 増やす。これをすべての学生について行う。最後に、c を先頭から調べて 2 以上になっている値の和を求める。

第6章　数値計算

コンピュータに数学的な問題を解かせることは多い。例として $f(x) = \dfrac{x}{(x+1)(x+2)}$ という関数が与えられたときに、$f(x)$ を $x=0$ から $x=1$ まで積分した $\displaystyle\int_0^1 f(x)dx$ の値を求める問題を考えてみよう。

人間がこれを解く場合、$f(x)$ の積分形を求めた上で、その式の計算をする。つまり、$\displaystyle\int f(x)dx = 2\log(x+2) - \log(x+1)$ $(= g(x)$ とおく$)$ と解いて、$\displaystyle\int_0^1 f(x)dx = g(1) - g(0) = \log(9/8)$ のように答を得る方法である。

コンピュータを使っても、同じように微分・積分や代数演算の規則を使って問題を解かせることができる。このような解き方は**数式処理**と呼ばれ、現在では学校で習う程度の数学の問題ならば自動的に解けるようになっている。下の画面は Mathematica という数式処理ソフトウェアで例の問題を解かせたものである。(In と書かれた行が入力した式で、Out と書かれた行がその答である。)

```
In[54]:= f[x_] = x / ((x + 1) (x + 2))

Out[54]= x / ((1 + x) (2 + x))

In[56]:= g[x_] = Integrate[f[x], x]

Out[56]= -Log[1 + x] + 2 Log[2 + x]

In[57]:= g[1] - g[0]

Out[57]= -3 Log[2] + 2 Log[3]
```

関数 f を定義し

その積分を求め g とする

求められた式

ただし、問題によっては数式処理をした結果が簡単な式では表わせないような場合があるので、この方法はいつも使えるとは限らない。例えば Mathematica では $\displaystyle\int \frac{\sin x}{\log x} dx$ の答を求めることはできない。

```
In[60]:= h[x_] = Integrate[Sin[x] / Log[x], x]

Out[60]= ∫ Sin[x] / Log[x] dx
```

(Out と書かれた行が積分記号を使った式のままになっているのは、積分形を見つけられなかったことを意味している。)

自然科学や工学における数学的問題は、数式処理では解けないことがめずらしくないため、反復計算によって近似的な解を求める**数値計算**と呼ばれる方法がとられることが多い。この章では数値計算によって問題を解く方法をいくつか紹介し、その際に必ず生じる誤差の問題についても触れる。

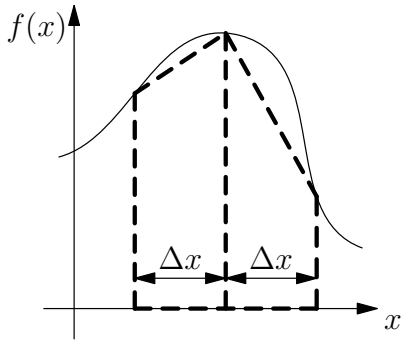
6.1　数値積分

まずは数値計算による積分から見てゆこう。関数 $f(x)$ が与えられたときにその定積分 $\int_{x_s}^{x_e} f(x)dx$ を求める問題である。

$f(x)$ のある区間の積分値を求めるということは、$f(x)$ のグラフを描き、グラフによって囲まれる領域の面積を求めることに相当する。$f(x)$ を実際に計算することによってグラフの高さを求めることはできるので、これを使って近似的な面積を求めるのが数値積分の方法である。この際、近似のやり方には**台形公式**や**シンプソン公式**などと呼ばれる方法が知られている。

6.1.1　台形公式

台形公式は、下図のように積分する範囲を幅 δx の区間に分け、各区間の領域を台形に近似して面積を求める方法である。



別の言い方をすると、関数を $f(x)$ を一定間隔の折れ線によって近似て、その関数を積分していることになる。

x_s から x_e までの積分範囲を n 等分した場合の近似式は次のようになる。まず、区間の幅を $\Delta x = \dfrac{x_e - x_s}{n}$ とおく。(左端を 0 番目とした)i 番目の区間の x 座標は $x_s + i\Delta x$ から $x_s + (i+1)\Delta x$ までであるので、この区間の台形の面積は

$$\frac{1}{2}\left\{f(x_s + i\Delta x) + f(x_s + (i+1)\Delta x)\right\}\Delta x$$

である。従って、積分値の近似値は x_s から x_e までの合計

$$\int_{x_s}^{x_e} f(x)dx \quad \simeq \quad \sum_{i=0}^{n-1} \frac{1}{2}\left\{f(x_s + i\Delta x) + f(x_s + (i+1)\Delta x)\right\}\Delta x \quad (6.1)$$

$$= \Delta x \left\{ \frac{f(x_s)+f(x_e)}{2} + \sum_{i=1}^{n-1} f(x_s + i\Delta x) \right\} \qquad (6.2)$$

となる。

式 6.2 を単純な繰り返しによって計算するのが以下の関数 `trapezoid(xs,xe,n)` である。`xs`, `xe` は積分範囲の両端の x の値を、`n` は分割数である。またこのプログラムでは、積分される関数として $f(x) = \dfrac{x}{(x+1)(x+2)}$ も同時に定義している。

```
1  def f(x)
2    x/((x+1.0)*(x+2.0))
3  end
4
5  def trapezoid(xs,xe,n)
6    deltax = (xe-xs)*1.0/n
7    sum = (f(xs)+f(xe))/2.0
8    for i in 1..(n-1)
9      sum = sum + f(xs+i*deltax)
10   end
11   deltax * sum
12 end
```

ファイル 6.1: `trapezoid.rb`

以下は、0 から 1 までの範囲を 100 分割して近似した場合の画面である。

```
1  irb(main):005:0> trapezoid(0,1,100)
2  => 0.11777910054096
3  irb(main):006:0> def g(x)
4  irb(main):007:1>   2*log(2+x)-log(1+x)
5  irb(main):008:1> end
6  => nil
7  irb(main):009:0> g(1)-g(0)
8  => 0.117783035656384
```

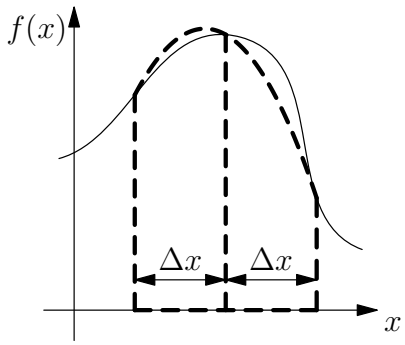
3 行目以下は、解析的に積分した $g(x) = \displaystyle\int f(x)dx = 2\log(x+2) - \log(x+1)$ を使って積分値を求めている。`trapezoid` の答が小数点以下 4 桁まで一致していることが分かる。

練習 6.1 (台形公式の正確さ) `trapezoid(0,1,n)` と $g(1)-g(0)$ の計算値は小数点以下何桁まで一致しているかを調べよ。8 桁目まで一致させるために

は n はどのくらい大きくなければいけないか、n = 100, 1000, 10000, . . . と変化させて調べよ。積分区間が $x = -0.9$ から $x = 0$ までの場合はどうか。

6.1.2　Simpson 公式

台形公式はグラフを折れ線、つまり δx の区間を直線で近似していた。Simpson 公式は直線のかわりに 2 次曲線で近似するものである。



x_s から x_e までの積分範囲を n 等分した場合の近似式は次のようになる。2 次式での近似のために 3 点の値を用いるので、近似する区間の**半分**の幅を $\Delta x = \dfrac{x_e - x_s}{2n}$ とおく。左端を 0 番目とした i 番目の区間の x 座標は $x_s + 2i\Delta x$ から $x_s + (2i+2)\Delta x$ までであり、さらに区間の中央の $x = (2i+1)\Delta x$ における値を用いて 2 次式を作り、それの積分を求めるとこの区間の面積は

$$\frac{1}{3}\left\{f(x_s + 2i\Delta x) + 4f(x + (2i+1)\Delta x) + f(x_s + (2i+2)\Delta x)\right\}\Delta x \quad (6.3)$$

となる。(この式の導出は後で述べる。) 従って積分値の近似値は x_s から x_e までの合計

$$\begin{aligned}
&\int_{x_s}^{x_e} f(x)dx \\
&\simeq \sum_{i=0}^{n-1} \frac{1}{3}\left\{ \begin{array}{r} f(x_s + 2i\Delta x) + 4f(x_s + (2i+1)\Delta x) \\ +f(x_s + (2i+2)\Delta x) \end{array} \right\}\Delta x \\
&= \frac{\Delta x}{3}\left\{ \begin{array}{l} f(x_s) + 4f(x_s + \Delta x) + f(x_e) \\ +\displaystyle\sum_{i=1}^{n-1}\left(2f(x_s + 2i\Delta x) + 4f(x_s + (2i+1)\Delta x)\right) \end{array} \right\} \quad (6.4)
\end{aligned}$$

となる。

練習 6.2 (Simpson 公式による積分)　a) 式 6.4 に従って積分を行う関数 `simpson(xs,xe,n)` を定義せよ。(注意: 台形公式の場合と違って Δx が積分範囲の $1/2n$ になっている。)

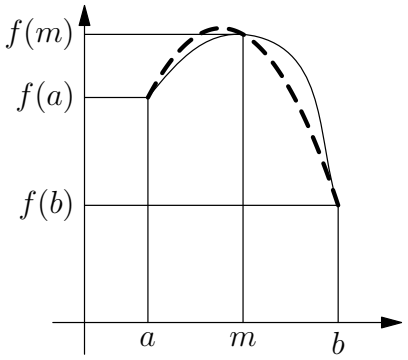
b) 練習 6.1 と同様に `simpson(xs,xe,n)` の解析的な積分値との違いを調べよ。

```
irb(main):011:0> simpson(0,1,100)
=> 0.117783035638943
```

Lagrange 補間と Simpson 公式 *

式 6.3 の導出方法を説明する。いま、区間の両端を a, b, 区間中の 1 点を m だとする。$f(a), f(m), f(b)$ の 3 点を通るような Lagrange 補間式 $P(x)$ は次のような 2 次関数になる。

$$\begin{aligned} P(x) &= \frac{(x-b)(x-m)}{(a-b)(a-m)}f(a) + \frac{(x-a)(x-b)}{(m-a)(m-b)}f(m) \\ &\quad + \frac{(x-a)(x-m)}{(b-a)(b-m)}f(b) \end{aligned} \tag{6.5}$$



ここで m を a, b のちょうど中間、つまり $m = \dfrac{b+a}{2}$ とした上で $P(x)$ を $x = a$ から b まで積分すると

$$\int_a^b P(x)dx = \frac{b-a}{6}\{f(a) + 4f(m) + f(b)\}$$

となり、$a = x, \quad b = x + 2\Delta x, \quad m = x + \Delta x$ とおくと

$$\int_a^b P(x)dx = \frac{1}{3}\{f(x) + 4f(x+\Delta x) + f(x+2\Delta x)\}\Delta x$$

を得る。

6.1.3　Monte Carlo 法による積分

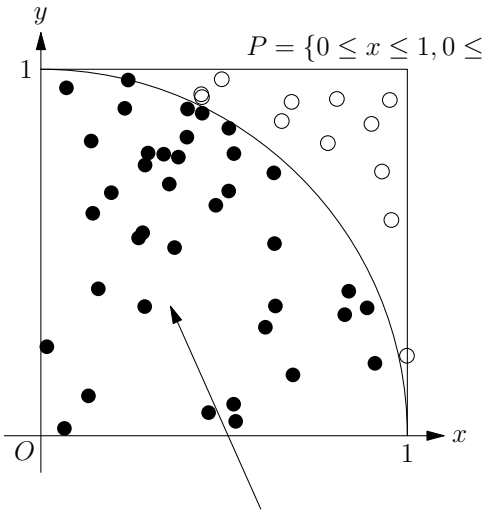
積分される関数が高次元の場合には Monte Carlo 法を用いた数値積分を行うことがある。

ここでは簡単のために 1 次元の関数を積分して四分円の面積を求める問題で説明する。つまり、

$$f(x) = \sqrt{1 - x^2}$$

を $x = 0$ から 1 まで積分することを考える。

積分範囲における $f(x)$ のグラフは下図のようになっている。



(黒い小円が $y < f(x)$ を満たしているものである)

まず、積分範囲の $f(x)$ を含むような長方形の領域

$$P = \{0 \le x \le 1, 0 \le y \le 1\}$$

に n 個の点をデタラメに配置する (図の黒い小円と白い小円)。次にこれらの点の中で、$y < f(x)$ となっているもの、つまり (図の黒い小円) の数 m を数える。この場合は領域

$$Q = \{x^2 + y^2 < 1, 0 \le x \le 1, 0 \le y \le 1\}$$

に含まれているものを数えればよいので $x^2 + y^2 < 1$ かどうかで判断すればよい。

デタラメに配置した点が均等に分布している場合、P, Q にそれぞれ含まれている点の数 n, m は P, Q の面積に比例する。P の面積は 1 であるので、Q の面積、つまり積分値は

$$\int_0^1 f(x)dx = Q \simeq \frac{m}{n}P = \frac{m}{n}$$

によって求められる。

この方法による積分を行う関数 `montecarlo(n)` をファイル 6.2 に示す。4 行目とその次の行で用いられている関数 `rand()` は、0 以上 1 未満の乱数を 1 つ発生させるものである。

```
def montecarlo(n)
  m=0
  for i in 1..n
    x=rand() # random number in [0,1)
    y=rand()
    if x*x + y*y < 1.0
      m = m + 1
    end
  end
  m*1.0/n
end
```

ファイル 6.2: montecarlo.rb

練習 6.3 (Monte Carlo 法の正確さ) Monte Carlo 法で高い精度の結果を得るためには、非常に多くの点を用いる必要がある。Q の面積は半径 1 の円の面積の 1/4 になることから montecarlo(n) の結果と解析的に求めた積分値を比較し、n を $10, 100, 1000, \ldots$ と増やした場合に何桁目まで一致するかを調べよ。(注意: 乱数を用いているので n の大きさに関らず偶然良い結果や悪い結果になることがあるので、同じ n の値で何回か実行してみよ。)

6.2　乱数と Monte Carlo 法

関数 montecarlo で使われていた rand() は**乱数**を発生させる関数であった。人間であればサイコロを振ることで乱数を作ることができるが、コンピュータは、基本的には与えられたデータを用いた計算しか行わないために、乱数を作ることは意外に難しい。

6.2.1　乱数とは何か

そこで、まず乱数の定義を与えた上で、コンピュータが作る乱数についての説明をしておこう。

まず定義であるが、ある 1 つの数だけを見てそれがデタラメな数かどうかを言うことはできない。乱数とは、数を何個も作ったときに、それらがデタラメになっているかどうかによって定義される。つまり数列、**乱数列**として定義される。より正確には次のように定義される。

次のような数列

$$\{x_1, x_2, \ldots, x_n, x_{n+1}, \ldots\}$$

の x_{n+1} の値が、$x_1, x_2, \ldots, x_n$ からは予測できず、x_i 全体の値はある確率分布に従うとき、この数列は**乱数列**であるという

練習 6.4 (乱数列となる条件) 次のような数列は乱数列と言えるかどうか考えよ。

a) ある地点の毎日の最高気温を日付順に並べたもの

b) 毎年の年末ジャンボ宝くじの当籤番号を年ごとに並べたもの (抽籤方法は数字が書かれた回転する円盤を矢で射て決めており、数字の範囲と当籤本数は毎年同じだとする)

c) 52 枚のカードを充分にシャッフルした後で一列に並べ、各カードの数字を取り出したもの

6.2.2　乱数と確率分布

代表的な確率分布の一つは**一様分布**で、この分布に従う乱数は**一様乱数**と呼ばれる。一様乱数は、出現確率が値によらず一定になっている。例えば、サイコロを振って出た数を並べたものは、前後の数と関係がないため乱数列になる。そして 1 から 6 までの数字が 1/6 の確率で出現するため一様乱数である。

次に放射性原子の集まりがあったときに、一定時間あたりに崩壊する原子の個数を数え、それを並べた数列を考えてみよう。1 つ 1 つの原子核が、他の原子と無関係に崩壊するのであれば、これは乱数列になる。しかし、その個数は平均的には一定の値になるが、ある個数になる確率は個数によって異なる。つまり、これは一様ではない乱数列になっている。

値の出現確率が正規分布に従うような乱数は**正規乱数**という。正規分布に従うとは、確率変数 x が ξ よりも小さい値をとる確率 $P(x)$ が以下の正規分布に従う場合である。

$$P(x < \xi) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\xi} \exp\left(-\frac{x^2}{2}\right) dx$$

自然現象における確率は上の放射性原子の例のように正規分布に従うものも多いため、自然現象を扱うシミュレーションでは、一様乱数と共に正規乱数をよく用いる。

6.2.3　疑似乱数列

さて、Monte Carlo 法のところで用いた関数 `rand()` は「乱数を発生させる」と説明されていたが、実際には何をどのようにして作っているのだろうか。

この関数は、乱数列**のように見える**数列を計算によって求めている。n 個目までの乱数列が作られているときに、$n+1$ 個目の数を求める計算は、直前の m 個の値 $x_n, x_{n+1}, \ldots, x_{n-m+1}$ を用いた、以下のような式で表わされる。

$$x_{n+1} = f(x_n, x_{n+1}, \ldots, x_{n-m+1})$$

つまりこの数列の値は、その値より前の値と無関係だとは言えない (それどころか前の値を使って作られている!) ので、本当の意味での乱数列とは言えず、**疑似乱数列**と呼ばれる。

また、それぞれの値は有限の桁数で表わされる数であるので、沢山乱数を作っていると、必ず同じ値の繰り返しに戻ってしまう。従って、疑似乱数列を作る場合は、同じ値の繰り返しに戻るまでの個数、つまり疑似乱数の**周期**ができるだけ大きくなるように設計することが望ましい。疑似乱数を作るアルゴリズムを設計する場合は、一様分布に関する適合度、統計的独立性などを様々な乱数検定法で検定し疑似乱数としての資格を与えている。

6.2.4　Monte Carlo 法

第 6.1.3 節では、乱数を使って積分を求める方法を Monte Carlo 法として紹介したが、Monte Carlo 法は乱数を用いた数学的問題の解法の総称であり、積分以外の問題にも用いられる。

Monte Carlo 法がよく用いられる問題には**決定論的問題**と**非決定論的問題**の 2 種類がある。

決定論的問題は、厳密な解が存在する問題であり、乱数によって近似的に求めるものであり、(多次元の) 積分が代表例である。

非決定論的問題は、自然科学や社会科学におけるシミュレーションのように、問題の中に確率的な変動を含むような問題を指す。このような問題の多くは実際に実験が困難であったり多大な費用がかかったりするので、乱数を用いて疑似的な実験を行うことで問題を「解く」わけである。このような問題の例としては、確率的に移動する粒子が時間経過とともにどこに移るかを調べるようなランダムウォークがある。

6.3　実数データ表現と誤差

数値計算では実数値を用いる場合が多い。コンピュータ上では、実数といっても有限の精度でしか表現できないため、数値計算の結果は真の値の近似値になってしまう。計算結果がどのくらい正しいかを判断するためには、数値がどのように表現されているかを知ることと、どのような場合に誤差が生じるかを知っておく必要がある。

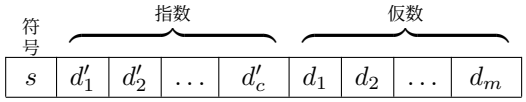
6.3.1　実数データ表現

コンピュータ上での数値データは 0 と 1 を並べた**ビット列**によって表わされる。数値計算で用いられる数値データは、桁数が非常に大きい数から 0 に近い小さな数までが同時に用いられる一方で、高速に計算することも求められているため、**浮動小数点表現**が用いられる。この浮動小数点表現は、数値を**符号部**、**指数部**、**仮数部**の 3 つに分けて表わす方法である。

浮動小数点表現にも色々な表現方法があるが、現在のコンピュータではIEEE 754 という規格が多く用いられている。

> 注意:　以下では、2 進数で表わされた数を使うことがあるので、そのような場合は区別のために数の後ろに $_{(2)}$ を付ける。10 進数であることを明記する場合は $_{(10)}$ を付ける。例えば $1101_{(2)}$ は 2 進数であり、10 進数で書くと $13_{(10)}$ という数を表わしている。また、$x_1x_2\ldots x_{n(2)}$ のように書いた場合は、上から i 桁目が x_i であるような n 桁の 2 進数を意味する。

IEEE 754 規格の中にもいくつかの表現方法があるのだが、その中でも現在よく用いられているものは、32 ビットのビット列によって表わす**単精度**表現と、64 ビットのビット列によって表わす**倍精度**表現である。これらは、32 または 64 ビットのビット列を



という 3 つの部分に分け、

$$(-1)^{s(2)}1.d_1d_2\ldots d_{m(2)} \times 2^{d'_1d'_2\ldots d'_{c(2)}-b}$$

という数を表わす方法である。

指数部のビット数 (c), 仮数部のビット数 (m) および (後述する) バイアス値 (b) は単精度表現か倍精度表現かによって異なり、以下のように決められている。

	仮数部 m	指数部 c	バイアス値 b
単精度	23	8	127
倍精度	52	11	1023

例えば次のような単精度 (32 ビット) の浮動小数点数があった場合、

0	1	0	0	1	1	0	1	1	0	0	0	1	1	1	0	1	1	1	1	0	0	1	1	1	1	0	0	0	0	1	0

符号 は 0 ビット目を $s = 0$ として (-1) の s 乗、つまり 0 ならば正、1 ならば負、

指数 は 1 から 8 ビット目を正の整数として、そこからバイアス値 b を引いた値、つまり $d'_1 \dots d'_{8(2)} - b = 10011011_{(2)} - b = 155 - 127_{(10)} = 28_{(10)}$

仮数 は 9 ビット目から 31 ビット目までを 2^{-1} から 2^{-23} に対応させ、さらに 1 を足した値、つまり $1.d_1 \dots d_{23(2)} = 1.00011101111001111000010_{(2)} = 2^0 + 2^{-4} + 2^{-5} + 2^{-6} + \cdots +_{(10)} = 1.116814_{(10)}$

であるので、

$$(-1)^0 \times 1.00011101111001111000010_{(2)} \times 2^{10011011_{(2)}-127}_{(10)} = 1.116814 \times 2^{28} = 2.997925 \times 10^8$$

という数を表わしている。

ただし、この表現だけでは 0 を表わすことができない。そこで、$d'_1 \dots d'_c$ と $d_1 \dots d_m$ が全て 0 の場合は 0 を表わす、などの特別な決まりも定められている。0 の他にも、無限大や計算結果が実数でなくなったことを表わす特別な値もある。

練習 6.5 (浮動小数点数の表現)　a) 次のビット列は IEEE 754 規格の単精度浮動小数点数である。どのような値を表わしているかを求めて 10 進数で書け。

$$\overset{s}{1}\overbrace{11111110}^{d'_1 \dots d'_8}\overbrace{11000000000000000000000}^{d_1 \dots d_{23}}$$

b) 次の数を IEEE 754 規格の単精度浮動小数点数で表わすときの $s, d'_1 \dots d'_8, d_1 \dots d_{23}$ をビット列で書け。

(a) 1.0

(b) $3145728_{(10)}$

(c) $5/65536_{(10)}$ ($\simeq 7.62939453125 \times 10^{-5}$)

練習 6.6 (浮動小数点の大小 *) 2 つの数 x, y が IEEE 754 規格によって 32 あるいは 64 ビットの浮動小数点数として表わされていたとする。このビット列を 32 あるいは 64 ビットの整数だと解釈すると、全く別の数を表わしていることになるが、x と y の間の大小関係だけは保たれる。このことを証明せよ。ただし、整数と解釈する場合は 2 の補数表現が用いられているものとする。

6.3.2　丸め誤差

浮動小数点数を用いた数値計算は、一定の桁数を使った近似値を使って計算をするため、正確な値からの誤差が生じる。多くの場合、この誤差は無視

できるほど小さいのだが、場合によっては計算の正確さを損なうほど大きくなる。そのため、どのような場合にどのような誤差が生じるかを知っておくことは数値計算によって意味のある結果を得るためにも重要である。

最初に紹介するのは**丸め誤差**というものである。これは 10 進数だと正確に表わせる数でも有限桁の 2 進数では近似値になってしまうことで生じる誤差である。

例えば有限桁の 10 進数で 1/3 を表わすことを考えてみよう。1/3 は $0.33333\cdots$ という無限に続く小数なので、小数点以下 5 桁目で切り捨てた近似値 0.33333 は正しい値 1/3 よりも $0.0000033333\cdots$ だけ小さい値になってしまう。

2 進数を使っている場合に気を付けなければいけないのは、10 進数では有限の桁数で正確に表わされているのに、2 進数で表わすと無限に続く小数になってしまうような数である。例えば 1/10 は 10 進数では 0.1 という有限の桁数で正確に表わせる。しかし 2 進数にすると、

$$
\begin{aligned}
0.1_{(10)} &= 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + \cdots_{(10)} && (6.6)\\
&= 0.0001100110011\cdots_{(2)} && (6.7)\\
&= 1.100110011\cdots_{(2)} \times 2^{-4} && (6.8)
\end{aligned}
$$

という循環小数になってしまう。

これを IEEE 754 規格の倍精度浮動小数点数として表わす場合、仮数部は (先頭の 1 を除いた)52 桁で表現するので、以下の下線部分が使われる。

$$1.\underline{1001100110011001100110011001100110011001100110011001}1001\cdots$$

さらに、切り捨てられる桁は 0 捨 1 入 (0 ならばそのまま、1 ならば切り上げ) がされるので、

$$1.\underline{1001100110011001100110011001100110011001100110011010}$$

という数に近似される。最後の 2 桁が切り上げによって変化していることに注意してほしい。そのため、この数は 0.1 よりもわずかに大きな値を表わしている。実際、Ruby でも以下の計算によって確認できる。

```
1 irb(main):003:0> 0.1 * 3 == 0.3
2 => false
```

$0.3_{(10)}$ は 2 進数で表わすと $1.001100110011\cdots \times 2^{-2}$ という循環小数である。どちらも循環小数であるのに値が一致しないことは、$0.1_{(10)} \times 3_{(10)}$ が 2 進数でどのように計算されるかを追うことで確認できる。この計算は以下のように進む。

$$
\begin{array}{rl}
1.10011001100\ldots110011010 & {}_{(2)} \times 2^{-4}_{(10)} \\
\times) \qquad\qquad 1.1 & {}_{(2)} \times 2^{1}_{(10)} \\
\hline
110011001100\ldots110011010 & {}_{(2)} \times 2^{-3}_{(10)} \\
+)\ 110011001100\ldots110011010 & {}_{(2)} \times 2^{-3}_{(10)} \\
\hline
10.011001100110\ldots011001110 & {}_{(2)} \times 2^{-3}_{(10)} \quad (*) \\
= 1.\underbrace{0011001100110\ldots0110100}_{52\text{ 桁}} \hookleftarrow & {}_{(2)} \times 2^{-2}_{(10)}
\end{array}
$$

(*) の行までの計算は正確に行われるが、この値は先頭の 1 を除いて 54 桁あるので、末尾の 2 桁が 0 捨 1 入される。結局、繰り上がりが起きて最後の行のような値になる。

一方、$0.3_{(10)}$ の 2 進数表現は $1.001100110011\cdots \times 2^{-2}$ だったので、52 桁の仮数部は ...0110011 で終わる。よって、$0.1_{(10)} \times 3_{(10)}$ の方が

$$0.\underbrace{00\ldots001}_{52\text{ 桁}}{}_{(2)} \times 2^{-2} = 2^{-54}$$

だけ大きくなってしまっている。

このことも、次のような計算式で実際に確かめられる。

```
1 irb(main):005:0> 0.1 * 3 - 0.3
2 => 5.55111512312578e-17
3 irb(main):006:0> 2 ** -54
4 => 5.55111512312578e-17
```

練習 6.7 (丸め誤差) irb で `0.1 * 5 - 0.5` を計算し、その結果を説明せよ。

6.3.3　浮動小数点数の精度

ここで実数データ表現が扱える数値の範囲を整理しておこう。浮動小数点数表現の場合、仮数部の桁数は有効数字の桁数に相当する。また、指数部の桁数は表わされる数の大きさを決定する。

IEEE 754 規格の単精度表現の場合は仮数部が 23 ビット、指数部が 8 ビットであった。仮数部の先頭には 1 が付くので合計 24 ビットの情報を表わせるが、これは $\log_{10}(2^{24}) \simeq 7.2$ と約 7 桁の精度しかないことを意味している。

また、仮数部の表わせる値の範囲は $1.00\ldots0_{(2)} = 1$ から $1.11\ldots1_{(2)} \simeq 2_{(10)}$ までである。

指数部は $2^{-126} \simeq 1.2 \times 10^{-38}$ から $2^{127} \simeq 1.7 \times 10^{38}$ までの範囲の値をとれる。(指数部のビットがすべて 0 の場合とすべて 1 の場合は、0 や無限大などの特別な値を表わすために用いられるので除外してある。)

結局、単精度表現で表わせる最小の数は約 1.2×10^{-38}, 最大の数は $2\times1.7\times10^{38} \simeq 3.4 \times 10^{38}$ となる。

練習 6.8 (倍精度表現の精度) IEEE 754 規格の単精度表現の有効桁数と表現できる値の範囲は下のようにまとめられる。これにならって倍精度の場合の有効数と表現できる値の範囲を求めよ。

	単精度	倍精度
仮数部のビット数	23	52
10 進有効桁数 (約)	$\log_{10}(2^{23+1}) \simeq 7$	
指数部のビット数	8	11
最小の指数	$2^{-126} \simeq 1.2 \times 10^{-38}$	
最大の指数	$2^{127} \simeq 1.7 \times 10^{38}$	
最大値	$2 \times 1.7 \times 10^{38} \simeq 3.4 \times 10^{38}$	

6.3.4　桁落ち誤差

第 6.1 節で見た数値積分の方法は、微小区間の面積を合計して全体の面積を求めるというものであった。このときの区間の幅を短くしてゆけば、それだけ正確な値が求められるはずである。

このように微小な値を扱う場合は違った種類の誤差に注意しなければいけない。

例として $f(x) = \log(x)$ の微分を数値計算によって求める問題で説明しよう。$f(x)$ の導関数は $f'(x) = \lim_{h \to 0} \dfrac{f(x+h) - f(x)}{h}$ である。極限を実際に求めることはできないので、数値計算では適当に小さな h を使った近似値 $\dfrac{f(x+h) - f(x)}{h}$ を使う。理論的には h を小さくすれば小さくするほど正確な値になるはずであるが、実際にそうなるか確かめてみよう。

次に示すのは h を $1,\ 0.1,\ 0.01,\ 0.001,\ \ldots,\ (0.1)^{\texttt{h\_digits}}$ と 1 桁ずつ小さくしてゆきながら $\dfrac{f(x+h) - f(x)}{h}$ を求め、解析的に求めた $f'(x) = \dfrac{1}{x}$ との差を求めてゆく関数 `cancellation(x,h_digits)` の定義である。

h に対する誤差の変化を見るため、この関数では (先頭を 0 番目として) i 番目が $h = (0.1)^i$ のときの誤差であるような配列を作って答えている。

関数 `cancellation` を使って $x = 2$, $h = 1.0$ から 10^{-15} までについて誤差を求めた結果は次のようになる。

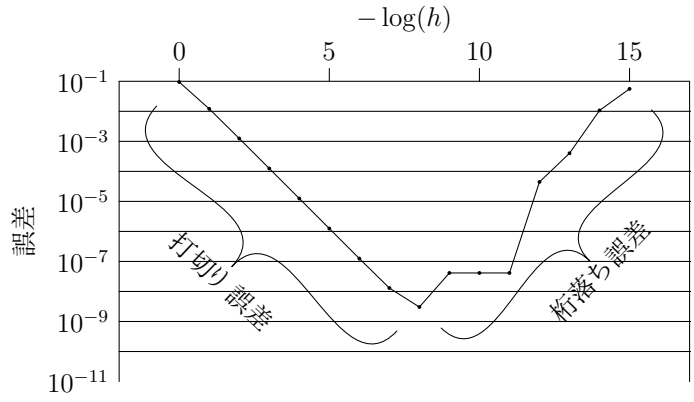
```
irb(main):004:0> cancellations(2.0, 15)
=> [0.0945348918918355, 0.0120983583056795,
    0.0012458488961028, 0.000124958348945658,
    1.24995825353524e-05, 1.24998572570423e-06,
    1.24941223755837e-07, 1.30307428736209e-08,
    3.03873576301683e-09, 4.13701852775006e-08,
    4.13701852775006e-08, 4.13701851664783e-08,
    4.44502911701727e-05, 0.00039963891867989,
    0.0107025913275716, 0.0559107901499378]
```

```
include(Math)
load("./abs.rb")

def f(x)
  log(x)
end

def cancellations(x,h_digits)
  errors=Array.new(h_digits+1)
  for i in 0..h_digits
    h = 0.1**i
    df=(f(x+h)-f(x))/h
    errors[i] = abs(df-1.0/x)
  end
  errors
end
```

ファイル 6.3: cancellation.rb (練習 3.1 で定義した abs を使用している)

最初は予想通り h を小さくするにつれて誤差が小さくなっている。この部分は粗い近似によって引き起こされる**打ち切り誤差**(後述) である。

しかし誤差は $h = 10^{-8}$ のときに約 3.0×10^{-9} となったのが最小で、その後は増加に転じている。横軸に h の桁数、縦軸に誤差の対数をとったグラフにすると下のようになる。



つまり h を小さくすればするほど、逆に誤差が大きくなっている。この後半の誤差は**桁落ち誤差**と呼ばれるものである。桁落ち誤差は、非常に近い 2 つの値の差を求めた場合に有効桁数が減ることで引き起こされる誤差を指す。

この例で言えば、h を小さくするにつれて $f(x+h)$ と $f(x)$ の上位桁はよく多く一致する。従って、その差の有効桁数はそれだけ減るため、誤差が生じている。

6.3.5　情報落ち誤差

数値積分では、微小領域を台形や 2 次関数で近似した面積を求め、それを足し合わせることで全体の面積を求めていた。微小領域の幅を小さくすればそれだけ近似が精確になるので、求められる解もより精確になるはずであるが、果たして本当だろうか。

そこで $1/n$ を n 回加えるという関数を定義し、n によってその答がどう変わるかを調べてみよう。これはつまり $f(x)=1$ という関数を 0 から 1 までの範囲を n 個に分けて数値積分することに相当する。ここでは $n=10^{\texttt{digits}}$ の場合に限って sum(digits) という関数を定義する。

```
1  # simulate 32 bit floating point representaiton
2  def coerce32(f)
3    [f].pack("f").unpack("f")[0]
4  end
5
6  def sum(digits)
7    n=10**digits
8    sum = 0.0
9    d = coerce32(1.0 / n)
10   for i in 1..n
11     sum = coerce32(sum + d)
12   end
13   sum
14 end
```

ファイル 6.4: lossofinformation.rb

この関数では、$1/n$ を計算するときと、合計する変数 sum に $1/n$ を加える際に coerce32 という関数を使っている。この関数は値を 32 ビット浮動小数点数に変換して、再び 64 ビット浮動小数点数に戻している。Ruby は (使用している環境にもよるが) 浮動小数点数には 64 ビット表現を用いているため、そのままだと情報落ち誤差の影響を観測することが難しいため、あえてビット数の少ない表現に変換している。

関数 sum(digits) を使って、$n=10^6, 10^7, 10^8$ についての値を求めた画面は以下のようになる。

注意:　$n = 10^8$ のときは 1 億回の繰り返しになるので、以下の計算には非常に時間がかかる。

```
irb(main):004:0> sum(6)
=> 1.0090389251709
irb(main):005:0> sum(7)
=> 1.06476747989655
irb(main):006:0> sum(8)
=> 0.25
```

$n = 10^6, 10^7$ のときの誤差は、丸め誤差が蓄積したものである。$n = 10^8$ のときに突然 0.25 という異常な値になっているが、これが情報落ち誤差である。

情報落ち誤差とは、大きな数に小さな数を足した場合に、小さな数が大きな数の有効桁の範囲外になることで無視されてしまうために起きる。この例では、$n = 10^8$ のときの d は 10^{-8} であり、sum が 1 に近い値の場合には 8 桁以上の有効桁がなければ d の値を足した分は無視されてしまう。32 ビット浮動小数点数の仮数部の有効桁数は約 7 桁なので途中から何回 d を加えても値が増えなくなってしまい、0.25 という結果になってしまっている。

6.3.6　打切り誤差

Taylor 展開のように、ある値が無限級数によって表わされているとき、その値を近似的に計算する場合には適当な数の項までを計算して以降の計算を打ち切ることになる。そのような場合、仮に計算を無限に高い精度で行っていたとしても打ち切られた以降の項が誤差として残ることになる。このような誤差を**打ち切り誤差**という。

例えば指数関数の原点の周りの Taylor 展開は、

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \cdots$$

という式であるが、この計算を無限回の計算を行うことはできないので、有限回 (n 回) で打切った近似値

$$e^x \simeq \sum_{i=0}^{n} \frac{x^i}{i!}$$

を計算することになる。このとき、打ち切り誤差の主要成分は

$$\frac{x^{n+1}}{(n+1)!}$$

となる。

6.4　連立 1 次方程式

$2x - y = 2$ と $-x + 2y = 5$ という方程式から x, y の値を求めるような連立 1 次方程式を n 個の変数に一般化した問題は、CT スキャナや検索エンジンにおける web ページの順位付けなどのように色々な場面で用いられる。

連立 1 次方程式の解法は、コンピュータの発明以前から数多くのものが考えられてきた。特に小規模な (つまり変数の数が少ない) 連立 1 次方程式は **Gauss 消去法**や **Gauss-Jordan 法**などの**消去法**によって解くことができる。これは人間が筆算で解く方法に相当する。

規模の大きな連立 1 次方程式は、**Jacobi 法**などの**反復法**という解法が用いられることが多いが、本書では扱わない。

6.4.1　Gauss 消去法

Gauss 消去法は、連立 1 次方程式の変数 1 つずつに注目し、その変数の係数を 0 にする (つまりその変数を消去する) ような変形を繰り返してゆく方法である。x, y, z からなる連立 1 次方程式であれば、1 つ目の方程式を使って他の 2 つの方程式から x を消去し、2 つの目の方程式を使って 3 つの目の方程式から y を消去して、z の値を求めるという手順になる。その後は、求めた z の値と 2 つの方程式から y の値を、そして y, z の値と 1 つ目の方程式から x の値を求めることができる。

実際の連立 1 次方程式でこの手順を説明しよう。連立 1 次方程式として次のようなものが与えられたとする。

$$\begin{array}{rcrcrcl} x & + & y & - & z & = & 2 \\ 3x & + & 5y & - & 7z & = & 0 \\ 2x & - & 3y & + & z & = & 5 \end{array}$$

行列演算を使うと、この方程式は次のような係数行列と変数ベクトル、定数ベクトルの間の方程式として書ける。

$$\underbrace{\begin{pmatrix} 1 & 1 & -1 \\ 3 & 5 & -7 \\ 2 & -3 & 1 \end{pmatrix}}_{\text{係数行列}} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \underbrace{\begin{pmatrix} 2 \\ 0 \\ 5 \end{pmatrix}}_{\text{定数ベクトル}}$$

消去法では、1 つの方程式を定数倍したり、ある方程式に別の方程式 (の定数倍したものを) 加えたりする操作を行う。これは行列と定数ベクトルの特定の行の定数倍や、行の間の加減算に相当するので、以降では係数行列と定数ベクトルをつなげた行列を使って考えることにする。変数ベクトルは変化し

ないので、説明のために下のような書き方を使う:

1	1	−1	2	r_1
3	5	−7	0	r_2
2	−3	1	5	r_3

これは左から係数行列、定数ベクトル、方程式の名前や操作を表わす。

さて、最初の行列・ベクトルが下のような状態だったとする。

1	1	−1	2	r_1
③	5	−7	0	r_2
2	−3	1	5	r_3

まずは r_1 を使って r_2, r_3 の (先頭を 1 として)1 列目の係数を 0 にする。③で示された係数を 0 にするためには r_2 から r_1 の 3 倍を引けばよい。この操作を $r_2 - 3r_1$ と表わすことにして、2 行目の右端に書くことにする。

同様に 2 で示された係数を消去するために $r_3 - 2r_1$ を行う。変形後の係数・定数は次のようになる。右端の列には、それぞれの行にどのような操作を行ったかが書かれている。

1	1	−1	2	r_1
0	2	−4	−6	$r_2 - 3r_1$
0	−5	3	1	$r_3 - 2r_1$

各方程式の名前は再び r_1, r_2, r_3 だとして、今度は〇で示された係数を調整する。つまり変形前が

1	1	−1	2	r_1
0	②	−4	−6	r_2
0	㊀5	3	1	r_3

であり、これの r_2 の 1 列目を 1 に、r_3 の 1 列目を 0 にするような変形を行う操作と結果は次のようになる。

1	1	−1	2	r_1
0	1	−2	−3	$r_2/2$
0	0	−7	−14	$r_3 + (5/2)r_2$

さらに 3 行 3 列を 1 にするような変形を行うと、次のようになる。

1	1	−1	2	r_1
0	1	−2	−3	r_2
0	0	1	2	$-r_3/7$

これで行列の対角成分が全て 1 であるような方程式が得られた。3 行目から逆順に z, y, x を求める**後退置換**を行うと、

$$\begin{aligned} z &= 2 \\ y &= 2z - 3 = 1 \\ x &= -y + z + 2 = 3 \end{aligned}$$

が最終的な解として求まる。

6.4.2　Gauss-Jordan 法

Gauss 消去法の最後は、消去によって得られた上三角行列に、後退置換を行っている。消去の手順を少し変更すると、係数行列を単位行列に変形することができ、後退置換を行う必要がなくなる。これが **Gauss-Jordan 法**である。

前出の連立 1 次方程式を用いて Gauss-Jordan 法の手順を見てゆこう。

1. まずは Gauss 消去法と同様に係数行列と定数ベクトルを並べたものから始める。

1	1	−1	2	r_1
3	5	−7	0	r_2
2	−3	1	5	r_3

2. r_1 を使って r_2, r_3 の第 1 列目の係数を消去する。これは Gauss 消去法と同一である。

1	1	−1	2	r_1
0	2	−4	−6	$r_2 - 3r_1$
0	−5	3	1	$r_3 - 2r_1$

3. 次に r_2 を使って他の行の係数を消去する。ここで Gauss-Jordan 法では r_1 の係数も消去する。

1	0	1	5	$r_1 - r_2/2$
0	1	−2	−3	$r_2/2$
0	0	−7	−14	$r_3 + (5/2)r_2$

4. さらに r_3 を使って r_1, r_2 の係数を消去する。

1	0	0	3	$r_1 + r_3/7$
0	1	0	1	$r_2 - 2r_3/7$
0	0	1	2	$-r_3/7$

このとき係数行列は単位行列になっているので、定数ベクトルの部分が与えられた連立 1 次方程式の解になっている。

Gauss-Jordan 法のプログラム

```
1  def gj(a) # Gauss-Jordan method without pivoting
2    row = a.length()
3    col = a[0].length()
4    for k in 0..(col-2)
5      akk = a[k][k]
6      for i in 0..(col-1)        # normalize row k
7        a[k][i]=a[k][i]*1.0/akk
8      end
9      for i in 0..(row-1)        # eliminate column k
10       if i != k                # of all rows but k
11         aik = a[i][k]
12         for j in k..(col-1)
13           a[i][j] = a[i][j] - aik * a[k][j]
14         end
15       end
16     end
17   end
18   a
19 end
```

ファイル 6.5: gj.rb

ファイル 6.5 に Gauss-Jordan 法によって n 変数の連立 1 次方程式を解く関数 gj(a) を示す。この関数は、係数行列と定数ベクトルを並べた n 行 $n+1$ 列の配列 a を引数として受け取り、行数と列数を変数 row, col に覚えておく。

関数全体は k 行目を使った消去を、0 行目から順に行ってゆく (4 行目からの繰り返し)。なお、Ruby では配列の番号は 0 から始まるので、以降では行や列は 0 行目から始まるとして説明する。

消去はまず、$\mathtt{a}_{kk}$ が 1 になるように k 行の係数・定数を正規化する。具体的には 6 行目からの繰り返しですべての列 i について $\mathtt{a}_{ki}$ を $1/\mathtt{a}_{kk}$ 倍している。このとき整数どうしの割り算によって切り捨てが起きること防ぐため、割り算に先立って 1.0 を先に掛けている。

次に 9 行目からの繰り返しで、k 以外のすべての行 i について係数 a_{ik} を消去する。そのためには、k 行を a_{ik} 倍した行ベクトルを i 行から引けばよいので、すべての列 j について a_{ij} から a_{kj} の a_{ik} 倍を引いている (13 行目)。

この繰り返しが終わると各行の最後の要素が、それぞれの変数の解になっている。関数 gj は、(単位行列へと変形された) 係数行列と定数ベクトルを

あわせた配列のまま答を返している。

```
irb(main):005:0> a=[[1, 1,-1,2],
irb(main):006:1*     [3, 5,-7,0],
irb(main):007:1*     [2,-3, 1,5]]
=> [[1, 1, -1, 2], [3, 5, -7, 0], [2, -3, 1, 5]]
irb(main):008:0> gj(a)
=> [[1.0, 0.0, 0.0, 3.0], [0.0, 1.0, 0.0, 1.0],
    [-0.0, -0.0, 1.0, 2.0]]
```

6.4.3　Pivoting 付き Gauss-Jordan 法

消去法では係数行列の要素で割り算を行う場所がある。関数 gj(ファイル 6.5) の 7 行目がそれである。このとき、

- a_{kk} が 0 の場合は、0 による割り算が行われてしまう、また、
- a_{kk} が 0 に近い値の場合は、その結果が非常に大きな値になり、その後で k 行目を a_{ik} 倍して i 行目から引く際に、大きな数どうしの引き算によって**情報落ち誤差**が生じてしまう

という問題がある。例えば

$$\begin{array}{rrrcr} x & -50y & -3z & = & -90 \\ -85x & +2y & -25z & = & -6 \\ 79x & +5y & +30z & = & -1 \end{array}$$

を解析的に解くと

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ -3 \end{pmatrix}$$

となるはずであるが、前出の gj(a) を使って解くと次のように誤差を含んだ結果になってしまう。

```
irb(main):010:0> b=[[  1,-50, -3,-90],
irb(main):011:1*     [-85,  2,-25, -6],
irb(main):012:1*     [ 79,  5, 30, -1]]
=> [[1, -50, -3, -90], [-85, 2, -25, -6], [79, 5,
    30, -1]]
irb(main):013:0> gj(b)
=> [[1.0, 0.0, 0.0, 0.999999999999977], [-0.0, 1.0,
     0.0, 2.0], [0.0, 0.0, 1.0, -2.99999999999996]]
```

このようなことを防ぐ方法として **pivoting** と呼ばれる行を入れ替えるやり方がある。具体的には、k 行目を使って消去する前に、$|a_{ik}|$ が最も大きい行 $i = p$ を選び、k 行と p 行を入れ替える。これは連立方程式の式の順序を入れ替えることに相当するため、自由に行うことができる。

$$
\begin{array}{rccccc}
 & \vdots & \vdots & \vdots & \\
\cdots & 1 & a_{k-1,k} & a_{k-1,k+1} & \cdots \\
k\text{ 行} \quad \cdots & 0 & \textcircled{$a_{k,k}$} & a_{k,k+1} & \cdots \\
\cdots & 0 & a_{k+1,k} & a_{k+1,k+1} & \cdots \\
 & \vdots & \vdots & \vdots & \\
i\text{ 行} \quad \cdots & 0 & a_{i,k} & a_{i,k+1} & \cdots \\
 & \vdots & \vdots & \vdots &
\end{array}
$$

Pivoting を行いながら Gauss-Jordan 法によって連立一次方程式を解く関数 `gjp(a)` をファイル 6.6 に示す。この関数は、ファイル 6.5 の `gj(a)` に 5 行目と 6 行目を追加しただけで、他はまったく同じものである。(ただし、そこで使われている `maxrow` の定義は省略してある。)

Pivoting は、k 行を使って消去を行う前に、`maxrow(a,k)` という関数を使って k 列目の係数の絶対値が最大となる行を k 行目以降から探し、その行番号を `max` にしまう (5 行目)。そして、`swap(a,k,max)` に配列 `a` の `k` 番目と `max` 番目を入れ替える、つまり行列の k 行を `maxrow` で見つけた行と交換する。

練習 6.9 (Pivoting の完成) 関数 `maxrow(a,k)` の定義を追加し、pivoting 付き Gauss-Jordan 法を完成させよ。さらに前出の連立 1 次方程式が、より小さな誤差で解けることを確認せよ。

```
irb(main):015:0> b=[[  1,-50, -3,-90],
irb(main):016:1*     [-85,  2,-25, -6],
irb(main):017:1*     [ 79,  5, 30, -1]]
=> [[1, -50, -3, -90], [-85, 2, -25, -6], [79, 5,
   30, -1]]
irb(main):018:0> maxrow(b,0)
=> 1
irb(main):019:0> maxrow(b,1)
=> 2
irb(main):020:0> gjp(b)
=> [[1.0, 0.0, 0.0, 1.0], [-0.0, 1.0, 0.0, 2.0],
   [0.0, 0.0, 1.0, -3.0]]
```

```
def gjp(a) # Gauss-Jordan method WITH pivoting
  row = a.length()
  col = a[0].length()
  for k in 0..(col-2)
    max=maxrow(a,k) # find absolute-maximal coeff.
    swap(a,k,max)   # swap rows
    akk = a[k][k]
    for i in 0..(col-1)       # normalize row k
      a[k][i]=a[k][i]*1.0/akk
    end
    for i in 0..(row-1)       # eliminate column k
      if i != k       # of all rows but k
        aik = a[i][k]
        for j in k..(col-1)
          a[i][j] = a[i][j] - aik * a[k][j]
        end
      end
    end
  end
  a
end
```

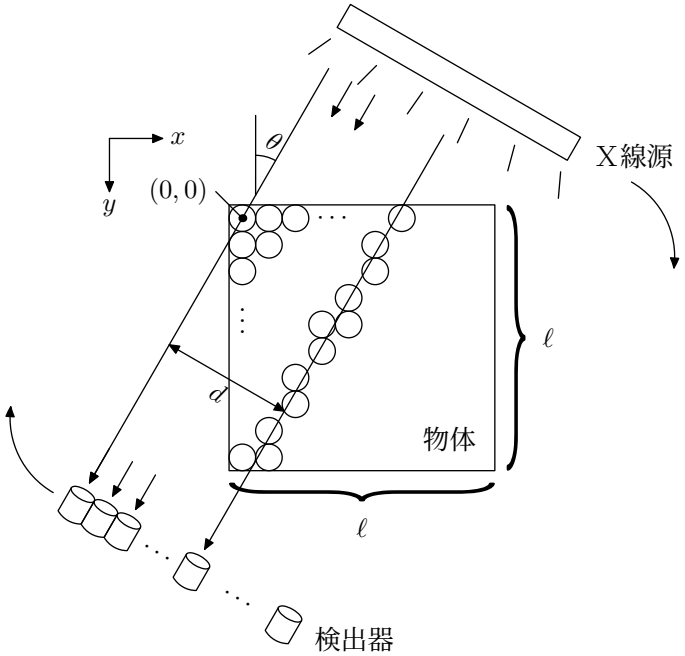
ファイル 6.6: gjp.rb (関数 gjp の定義のみ)

6.5　応用問題: CT スキャン*

6.5.1　原理

コンピュータ断層撮影 (computed tomography) とは、X 線装置を用いて人体の各方向から透過線量を測り、そのデータをもとに人体の横断面画像を逆算する技術である。

実際の CT スキャンでは X 線の拡散や計測誤差などの問題を扱わなければいけないが、ここではより簡単化した問題を考えることにしよう。ここでの CT スキャナは、次の断面図のように X 線源と検出器を並べたものが物体をはさんで回転しながら計測をする機器である。



物体は直径 1 の小さな円が $\ell \times \ell$ 個並べられて構成されているものとする。線源から照射された X 線は、物体の内部を通過して検出器に到達する。このとき X 線は物体内部の物質に吸収されるので、検出器は通過した線上の物質の密度の合計値を測ることになる。一列に並んだ検出器によって、異なる線上の物質についての情報が分かることになる。

これだけでは内部の構造は分からないので、CT スキャナ全体の角度 θ を何回か変えて同様の計測をする。すると各小円の密度を未知変数とした連立 1 次方程式を作ることができる。角度 θ のときに 1 つの検出器が得た値は、その検出器に到達する線上の小円の密度に関する 1 次方程式を 1 つ作るので、ℓ 個の検出器を使えば ℓ 個の連立 1 次方程式になる。角度を変えた測定を何回か行えば、解が存在するのに充分な数の連立 1 次方程式が得られる。

6.5.2　計算式

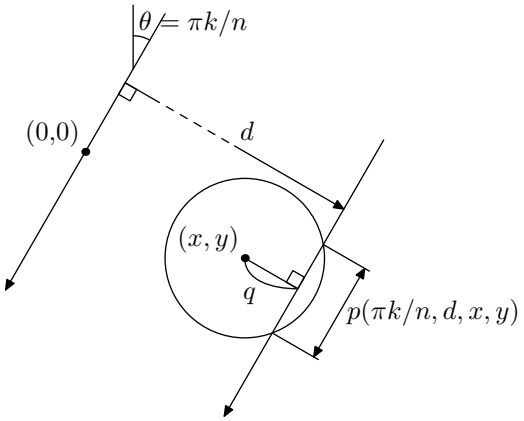
それでは計測値からどのような連立 1 次方程式を作ればよいのかを考えてゆこう。いま、検出器は間隔 1 で並び ℓ 個あり、端から $0, 1, 2, \ldots, \ell - 1$ という番号が付いているものとする。測定した n 回行い、そのときの角度 θ は $0, \pi/n, 2\pi/n, \ldots, \pi(n-1)/n$ だったとする。角度 $\pi k/n$ のときに d 番の検出器が測定した値を s_{kd} と書くことにする。また、左端の検出器に届く X 線は、常に左上の座標 (0,0) にある小円上を通過する位置にあるものとする。

座標 (x, y) にある小円の密度を変数 w_{xy} で表わすことにする。以下は、w_{xy} を未知数とした連立 1 次方程式を作ることになる。ここで s_{kd} として測定さ

れた X 線量は、その検出器に至る直線と、線上にあった各小円との交わりの長さと密度の積の和だとする。つまり、角度 $\pi k/n$ で原点との距離が d である直線と、中心が (x, y) にある小円との交わりの長さを $p(\pi k/n, d, x, y)$ で表わすことにすると、

$$s_{kd} = \sum_{\substack{x=0\ldots\ell-1 \\ y=0\ldots\ell-1}} p(\pi k/n, d, x, y) w_{xy} \tag{6.9}$$

ということである。ここで $p(\theta, d, u, v)$(ただし $\theta = \pi k/n$) は、下の図のような関係になっており、



$$\begin{aligned} p(\theta, d, u, v) &= \begin{cases} \sqrt{1-4q^2} & (q < \frac{1}{2}) \\ 0 & (q \geq \frac{1}{2}) \end{cases} \\ \text{ただし } q &= |x\cos\theta + y\sin\theta - d| \end{aligned} \tag{6.10}$$

となっている。

6.5.3　関数の定義

配布プログラム `scans.rb` には、関数 `scan1()`, `scan2()`, `scan3()`, `scan4()` が定義されている。これらの関数は、物体を CT スキャナによって測定された X 線量を n 行 ℓ 列の 2 次元配列 `s` を返す。行数 n(= `s.length()`) はスキャンの回数で、列数 ℓ(= `s[0].length()`) は画像の 1 辺の長さである。この配列の値 `s[`k`][`d`]` は、角度 $\pi k/n$ でスキャンしたときの d 番目の検出器の測定値である。

練習 6.10 (部分の通過距離) 式 6.10 に従って p(`theta,d,x,y`) を求める関数 `penetration(theta,d,x,y)` を定義せよ。

```
1 irb(main):004:0> penetration(0,0,0,0)
2 => 1.0
```

```
irb(main):005:0> penetration(3.141592/4,1,1,1)
=> 0.560096865715943
irb(main):006:0> penetration(3.141592/4,1,2,1)
=> 0.0
```

式 6.9 を $k = 0, \ldots, n-1,\ d = 0, \ldots, \ell - 1$ について連立させた 1 次方程式を係数行列 M、変数ベクトル $\mathbf{w}$、定数ベクトル $\mathbf{s}$ を用いて

$$M\mathbf{w} = \mathbf{s}$$

と書くことにする。ただし、$\mathbf{w}_{\ell y+x} = w_{xy}, \mathbf{s}_{\ell k+d} = s_{kd}$ である。このとき行列 M は

$$M_{\ell k+d,\ \ell y+x} = p(\pi k/n, d, x, y)$$

のように定められる。

練習 6.11 (係数行列の作成) 画像の 1 辺の長さが l, スキャンの回数が n のときの係数行列 M を作成する関数 coefficients(l,n) を定義せよ。

```
irb(main):008:0> coefficients(2,3)
=> [[1.0, 0.0, 1.0, 0.0], [0.0, 1.0, 0.0, 1.0],
    [1.0, 0.0, 0.0, 0.0], [0.0, 2.1073424255447e-08,
     0.963433044002285, 0.681250038633213], [1.0,
    2.98023223876953e-08, 0.0, 0.681250038633213],
    [0.0, 0.0, 0.963433044002285, 0.0]]
```

練習 6.12 (係数行列と定数ベクトルの合成) 係数行列 M と、測定値ベクトルを表わす配列 s から連立 1 次方程式を表わす配列を作る make_equations(l,n,m,s) を定義せよ。ただし l は 1 辺の長さ、n はスキャンの回数、m は $\mathtt{l} \times \mathtt{n}$ 行 $\mathtt{l}^2$ 列の配列、s は n 行 l 列の配列である。作られる配列は $\mathtt{l} \times \mathtt{n}$ 行 $\mathtt{l}^2 + 1$ 列の配列で、右端の列以外は m と同じ内容で、右端の列に s の内容を縦に並べた値が入る。

```
irb(main):010:0> make_equations(2,3,coefficients
    (2,3),[[1,1],[1.4,1.9],[1.4,1]])
=> [[1.0, 0.0, 1.0, 0.0, 1], [0.0, 1.0, 0.0, 1.0,
    1], [1.0, 0.0, 0.0, 0.0, 1.4], [0.0,
    2.1073424255447e-08, 0.963433044002285,
    0.681250038633213, 1.9], [1.0, 2.98023223876953e
    -08, 0.0, 0.681250038633213, 1.4], [0.0, 0.0,
    0.963433044002285, 0.0, 1]]
```

練習 6.13 (画像の復元) 測定値を表わす配列 s から 2 次元の画像を作成する関数 reconstruct(s) は次のように定義できる。この中で使われている equations_to_image(l,solved) は、配列 solved の (左端を 0 列目として)l^2 列目の上から l^2 個を $l \times l$ に並べ直した 2 次元配列を作るような関数である。この関数を定義して、reconstruct を完成させよ。

配布プログラム scans.rb に用意された関数 scan1(), scan2(), scan3(), scan4() が与える測定値から画像を作成せよ。(show(reconstruct(scan1())) のように実行すればよい。)

```
1 def reconstruct(s)
2   n=s.length()
3   l=s[0].length()
4   m=coefficients(l,n)
5   equations=make_equations(l,n,m,s)
6   solved=gjp(equations)
7   equations_to_image(l,solved)
8 end
```

ファイル 6.7: ctscan.rb 一部

6.6　章末問題

練習 6.14 (解析的な積分が難しい関数の数値積分) 関数 $g(x) = \dfrac{\sin x}{\log x}$ についての積分 $\displaystyle\int_{x_s}^{x_e} g(x)dx$ (ただし $x_e < 1$) について

a) 台形公式によって数値積分を求める関数 trapezoid_sinlog(xs,xe,n) を定義せよ。ただし n は分割の数とする。

b) Simpson 公式によって数値積分を求める関数 simpson_sinlog(xs,xe,n) を定義せよ。ただし n は分割の数とする。

```
1 irb(main):027:0> trapezoid_sinlog(0.1,0.9,100)
2 => -1.07129320182014
3 irb(main):028:0> simpson_sinlog(0.1,0.9,100)
4 => -1.07050038503067
```

練習 6.15 (Monte Carlo 法による 3 次元の積分) 半径 1 の球の体積を Monte Carlo 法によって求める関数 montecarlo3d(n) を定義せよ。ただし n は点

の個数だとする。半径 r の球の体積は $\frac{4}{3}\pi r^3$ であることが知られているので、これとの誤差を調べよ。

練習 6.16 (Monte Carlo 法による積分) 関数 $f(x) = \dfrac{x}{(x+1)(x+2)}$ を Monte Carlo 法によって数値積分する関数 `montecarlo_f(xs,xe,n)` を定義せよ。ただし、(`xs`, `xe`) は 0 < `xs` であるような積分区間、`n` は点の個数だとする。

ヒント: 区間 (`xs`, `xe`) において $f(x)$ が内側に収まるような長方形の領域を求め、その内側に点を打てばよい。$x > 0$ の範囲で $0 < f(x) < 3 - 2\sqrt{2}$ である。

第7章　パターン認識

2枚の顔写真が同じ人物かどうかを言い当てたり、人間の声から言葉を聞きとったりする処理では、沢山のデータの中に隠れている情報を見つけるような**パターン認識**が行われる。人間のパターン認識能力に対して、コンピュータが行うパターン認識は、膨大な計算時間がかかったり、不正確な結果しか得られないことも多い。そのため、コンピュータはパターン認識が不得意だと言われることも多い。

その一方で、コンピュータによるパターン認識の手法は年々進歩しており、今では様々な情報システムに応用されている。例えば指紋や顔写真などを大きなデータベースから検索することはすでに日常的に行われているし、手書き文字や音声によってコンピュータに文章を入力する方法も実用化の域に達している。これらは人間が行っているパターン認識をコンピュータに行わせる例だが、それ以外にも、DNAやタンパクの類似性を判別するような、人間が行うのも難しい問題に使われる場面も増えている。

この章では、パターン認識の特徴と、基本的なアルゴリズムを紹介する。また、その際に重要となるアルゴリズムとして**動的計画法**もあわせて説明する。

7.1　パターン認識の特徴

パターン認識の問題は、音や画像のようなデータの中に隠れているパターンを見つけることだと言える。例えば音声認識では、波形として得られた音声データの中から、母音や子音として知られている音素を見つけることをまず行う。このときは、例えば [æ] などの音素の波形データを用意しておいて、それが現われる場所を探すわけだが、実際の音声データはゆらぎや個人差、雑音があるために用意しておいたデータと完全に一致することはない。従ってパターン認識では、照合するデータと**似ているかどうか**、そしてどのくらい似ているかを判定しなければいけない。

実際の音声認識では波形を直接比較するのではなく、波形を周波数成分ごとの強弱に変換した上で比較を行う。

実際の情報システムでは、パターン認識を何段階も使うことが多い。音声認識であれば、最初に波形データから音素の並びを得るわけだが、次は音素の並びから音節や単語を作り、さらに単語の並びから文を作るという順に認識が行われる。このときも、各単語に対応した音素の並びを用意しておいて、それとの一致を調べることになるが、最初の段階で得た音素の並びには漏れや誤りがあるので、完全に一致することはやはりない。そのため、ここでも

ある程度の誤りを許して**似ている**音素データの並びになっているかを判定しなければいけない。

別の見方をすると、音素や単語に対して 1 つのデータを用意してそれとの一致を調べるかわりに、パターン認識では、音素や単語に対して「似ている」データの集まりを考え、実際のデータがその集まりに含まれるかどうかを判定していると言える。ただし、似たデータの集まりは有限集合とも限らず、また、どれだけ似ているかを比較することも必要となるので、パターン認識のアルゴリズムには様々な工夫が必要となってくる。

7.2　アラインメント

パターン認識の 1 つとして、2 つの文字列を、一致部分が対応するように並べる**アラインメント**問題をとりあげて、そのアルゴリズムを見てみよう。2 つの文字列の一致部分が多ければ多いほど似ている文字列ということになるので、例えば DNA の塩基配列の遺伝的な近さを調べる場合などに登場する問題である。

例えば GACGGATTAG と GATCGGAATAG という 2 つの文字列のアラインメントとは以下のようなものである。

0 1 2 3 4 5 6 7 8 9 0
GA_CGGA$\underline{\text{T}}$TAG
GA$\underline{\text{T}}$CGGA$\underline{\text{A}}$TAG
○ ○ × ○ ○ ○ ○ △ ○ ○ ○

このように並べると、上の文字列は下の文字列と比べて (左端を 0 文字目として)

- 2 文字目が欠けている (そこで**ギャップ**を挿入して位置を揃えてある)、
- 7 文字目が異なっている (**不一致**)、
- その以外の文字目は**一致**している

ことが分かる。文字列の下の記号はその位置が一致 (○)、ギャップ (×)、不一致 (△) のどれであるかを表わしている。

アラインメントはいくつも考えられる。上の例であれば、

0 1 2 3 4 5 6 7 8 9 0
GA$\underline{\text{C}}$_GGA$\underline{\text{T}}$TAG
GA$\underline{\text{T}}$$\underline{\text{C}}GGA\underline{\text{A}}$TAG
○ ○ △ × ○ ○ ○ △ ○ ○ ○

0 1 2 3 4 5 6 7 8 9 0
GA$\underline{\text{C}}$$\underline{\text{G}}G_A\underline{\text{T}}$TAG
GA$\underline{\text{T}}$$\underline{\text{C}}G\underline{\text{G}}A\underline{\text{A}}$TAG
○ ○ △ △ ○ × ○ △ ○ ○ ○

をはじめとして、沢山の可能性がある。

そこで通常は

一致 (○)	+2 点
ギャップ (×)	−2 点
不一致 (△)	−1 点

のように点数を決めておき、その総和が最も大きくなるものを最良のアラインメントとして採用する。(単にアラインメントと言った場合に最良のアラインメントを指すこともある。)

上に示した 3 つのアラインメントの得点は、順に 15, 12, 9 点である。さらに、他のアラインメントを検討しても 15 点以上のものはないことが分かるので、最初に示したものが最良のアラインメントとなる。

7.3　再帰によるアラインメント

アラインメントを求める方法を説明するために、まずは 2 つの文字列 $s =$ "ATAG", $t =$ "AAC" の最良のアラインメントの得点だけを求める問題を考えてみよう。

この問題は再帰的に考えることができる。"ATAG" と "AAC" のアラインメントは、後ろの文字から考えて

1. "ATAG" と "AA" のアラインメントに、ギャップと "C" を追加した場合: $\begin{array}{|c|}\hline \texttt{ATAG-} \\ \hline \texttt{AA C} \\ \hline \end{array}$ と書くことにする。
2. "ATA" と "AA" のアラインメントに、"G" と "C" を追加した場合: $\begin{array}{|c|}\hline \texttt{ATAG} \\ \hline \texttt{AA C} \\ \hline \end{array}$
3. "ATA" と "AAC" のアラインメントに、"G" とギャップを追加した場合: $\begin{array}{|c|}\hline \texttt{ATAG} \\ \hline \texttt{AAC-} \\ \hline \end{array}$

の 3 通りである。このとき、もし $\begin{array}{|c|}\hline \texttt{ATAG} \\ \hline \texttt{AA} \\ \hline \end{array}$, $\begin{array}{|c|}\hline \texttt{ATA} \\ \hline \texttt{AA} \\ \hline \end{array}$, $\begin{array}{|c|}\hline \texttt{ATA} \\ \hline \texttt{AAC} \\ \hline \end{array}$ の最良のアラインメントの得点がそれぞれ 0 点、2 点、0 点だったとすると、上の 1 から 3 の場合の得点は、それぞれに −2 点 (ギャップ) −1 点 (不一致)、−2 点を加えて −2 点、1 点、−2 点となる。従って最良のアラインメントは 2 の場合、つまり最後の文字どうしを「不一致」させるときで、得点は 1 点だと分かる。

$\begin{array}{|c|}\hline \texttt{ATAG} \\ \hline \texttt{AA} \\ \hline \end{array}$, $\begin{array}{|c|}\hline \texttt{ATA} \\ \hline \texttt{AA} \\ \hline \end{array}$, $\begin{array}{|c|}\hline \texttt{ATA} \\ \hline \texttt{AAC} \\ \hline \end{array}$ の最良のアラインメントの得点は、それぞれ再帰的に求めることができる。例えば、$\begin{array}{|c|}\hline \texttt{ATA} \\ \hline \texttt{AA} \\ \hline \end{array}$ であれば、$\begin{array}{|c|}\hline \texttt{ATA-} \\ \hline \texttt{A A} \\ \hline \end{array}$, $\begin{array}{|c|}\hline \texttt{ATA} \\ \hline \texttt{A A} \\ \hline \end{array}$, $\begin{array}{|c|}\hline \texttt{ATA} \\ \hline \texttt{AA-} \\ \hline \end{array}$ の 3 通りを考えて $-2-2=-4$ 点、$0+2=2$ 点、$1-2=-1$ 点となり、その中の最高点 2 点となる。

また、これを続けてゆくといずれ $\begin{array}{|c|}\hline \mathtt{AT} \\ \hline \\ \hline \end{array}$ のように、一方が空文字列となる場合の計算になる。このときはもう一方の文字列の長さだけギャップを入れるしかないので、$-2 \times 2 = -4$ 点のように計算できる。両方が同時に空文字列になる場合もあるが、その場合は 0 点である。

このやり方を一般化して文字列 s の先頭から i 文字と文字列 t の先頭から j 文字の最良のアラインメントの得点を求める関数 $a(i,j)$ は、次のような関係で表わすことができる。

$$a(i,j) = \begin{cases} G \times j & (i = 0) \\ G \times i & (j = 0) \\ \max \left(\begin{array}{l} a(i,j-1) + G, \\ a(i-1,j-1) + q(s[i-1], t[j-1]), \\ a(i-1,j) + G \end{array} \right) & (i > 0 \text{ かつ } j > 0) \end{cases} \tag{7.1}$$

ただし G はギャップの得点 (ここでは -2) を表わす定数、$q(x,y)$ は $x = y$ のときは一致の得点 $(+2)$、$x \neq y$ のときは不一致の得点 (-1) となる関数である。また $s[x]$ は、文字列 s の (先頭を 0 として)x 文字目を表わす。

これを Ruby の関数として定義するとファイル 7.1 のようになる。関数 $a(i,j)$ に対応するのは 15 行目から定義されている `align_sub(s,t,i,j)` である。この定義では、`i`, `j` のどちらかが 0 の場合の計算を 17 行目の式にまとめている。また、19 行目から 21 行目までは (1)`s` にギャップを挿入する場合、(2)`s` と `t` の文字を 1 つ一致または不一致させる場合、(3)`t` にギャップを挿入する場合、の 3 つの得点を計算している。22 行目の `max` を 2 回使った式はこれらの得点の最高点を求められている。

実際に計算させてみると、以下のようになる。

```
1 irb(main):005:0> align_rec("ATAG","AAC")
2 => 1
3 irb(main):006:0> align_rec("GACG","GCAG")
4 => 2
```

練習 7.1 (アラインメントの計算) “GACG” と “GCAG” の、得点が 2 点になるようなアラインメントを手で求めて書け。

また “ACGTA” と “AGCGA” の最良のアラインメントの点数を `align_rec` を用いて求め、その点数になるアラインメントを手で求めて書け。

練習 7.2 (再帰的アラインメントの計算量)　a) `align_rec` を用いて、5 文字の文字列どうしをアラインする時間を測れ。文字列の長さを 6,7,8 と変化させたとき、時間はどのように延びるか。

```
load("./max.rb")

def g()
  -2
end

def q(x,y)
  if x==y
    2
  else
    -1
  end
end

def align_sub(s,t,i,j)
  if i==0 || j==0
    i*g() + j*g()
  else
    v0 = align_sub(s,t,i,  j-1) + g()
    v1 = align_sub(s,t,i-1,j-1) + q(s[i-1],t[j-1])
    v2 = align_sub(s,t,i-1,j)   + g()
    max(v0,max(v1,v2))
  end
end

def align_rec(s,t)
  align_sub(s,t,s.length(),t.length())
end
```

ファイル 7.1: align_rec.rb

b) s が 100 文字の文字列だったとする。s にギャップを 10 個挿入する場合、そのやり方は何通りあるか。

c) s,t がともに 100 文字だったとする。それぞれにギャップを 10 個ずつ挿入して s,t をアラインする場合、アラインメントは何通りあるか。

7.4　動的計画法

再帰によるアラインメントは、文字列が長いときには時間がかかり過ぎて実用にならない。しかし、重複した計算を避けることでより長い文字列どうしのアライメントを現実的な時間で求めることができるようになる。この工夫は**動的計画法**とよばれ、アラインメントに限らず多くの問題を効率よく解くために用いられている。

まずはより簡単な問題を使って動的計画法の考え方を理解しておこう。プロ野球の日本シリーズでは、先に 4 勝したチームが優勝となる。チーム A,B がそれぞれあと i, j 勝すれば優勝するときに、A が優勝する確率を $P(i,j)$ と表わすことにする。1 回の試合で A が勝つ確率は $1/2$ だとすると、P は次のような再帰的な関係によって求めることができる。

$$P(i,j) = \begin{cases} 1 & (i = 0 \text{ かつ } j > 0) \\ 0 & (i > 0 \text{ かつ } i = 0) \\ \dfrac{P(i-1,j) + P(i,j-1)}{2} & (i > 0 \text{ かつ } j > 0) \end{cases} \tag{7.2}$$

これを再帰的に計算する場合、同じ (i,j) の計算を何度も行うことになる。

動的計画法は、このような計算をする際に、表を用意し、表の中身を順に埋めてゆくことで同じ計算の繰り返しを避けながら計算を行う方法である。

この場合であれば、$P(i,j)$ の値を記録する表を用意して、i, j の小さい方から値を埋めてゆく。(0 行目と 0 列目はそれぞれ 0 と 1 で先に埋めておくことができる。)

$j\backslash i$	0	1	2	3	4
0		0	0	0	0
1	1	1/2	1/4	1/8	
2	1	3/4	1/2		
3	1	7/8	□		
4	1				

上は途中まで表を埋めた様子である。ここで例えば $P(2,3)$ の値 (表で□となっている場所) は、$(P(1,3) + P(2,2))/2$ なので、表に書かれた値を使って $(7/8 + 1/2)/2 = 11/16$ となる。

求めたい値が $P(i,j)$ であれば、$(i+1) \times (j+1)$ の表を埋めればよいので計算量は $O(ij)$ となる。これは再帰的に計算する場合よりも圧倒的に小さい。

7.5　動的計画法によるアラインメント

7.5.1　最良のアラインメントを見つける

式 7.1 は、式 7.2 と似た性質を持っている。つまり、$a(i,j)$ を求めるために $a(i,j-1), a(i-1,j-1), a(i-1,j)$ を再帰的に求めているという点であ

る。また、$i = 0$ または $j = 0$ の場合は直接点数を求めることができる。

そこで、$a(i, j)$ の値を i 行 j 列に置いたような表 A を用意して、表を左上から右下に向かって埋めてゆけば、i 行 j 列の値 (A_{ij} と書く) は、その上、左、左上の 3 つの場所の値を使って求めることができる。長さが m, n であるような文字列 s, t の最良のアラインメントであれば、大きさ $(m+1) \times (n+1)$ の表を埋めることで A_{mn} の値を求められるので、計算は 2 つの文字列の長さの積に比例する時間で終わることになる。

下は $s =$ "ATAG", $t =$ "AAC" の場合に表を作っている様子である。

	t	A	A	C	
s	$i \backslash j$	0	1	2	3
A	0	0	−2	−4	−6
T	1	−2	2	0	2
A	2	−4	□		
G	3	−6			
	4	−8			

□となっている 2 行 1 列の値は、s を先頭から 2 文字 (AT)、t を先頭から 1 文字 (A) アラインさせる場合の得点になる。$s[1] \neq t[0]$ なので、得点は $\max(A_{11} - 2, A_{10} - 1, A_{21} - 2) = 0$ となる。

最後まで表を埋めると下のようになり、A_{43} の値 1 が s, t の最良のアラインメントの得点だと分かる。

	t	A	A	C	
s	$i \backslash j$	0	1	2	3
A	0	0	−2	−4	−6
T	1	−2	2	0	2
A	2	−4	0	1	−1
G	3	−6	−2	2	0
	4	−8	−4	0	1

練習 7.3 (表を用いたアラインメント) "GACG" と "GCAG" についてアラインメントを計算する表を書き、最良のアラインメントの点数を求めよ。

7.5.2　トレースバック

表 A が完成したら、表を右下から左上に逆向きに辿ることでギャップを挿入した文字列 u, v を作ることができる。$s =$ "ATAG", $t =$ "AAC" の場合であれ

ば下の表の 4 行 3 列から出発する。最初は u, v ともに空文字列である。

	t	A	A	C	
s	$i\backslash j$	0	1	2	3
A	0	0	−2	−4	−6
T	1	−2	2	0	2
A	2	−4	0	1	−1
G	3	−6	−2	2	0
	4	−8	−4	0	[1]

4 行 3 列の値を上、左上、左の値と比較する。すると、この値は左上から $A_{32}-1$ として求められたことが分かる。(左や上から求められた場合は −2 点になる。) つまり $s[3], t[2]$ を不一致とするのが最良の場合なので、$u =$ “G”, $v =$ “C” として 3 行 2 列に移動する。

	t	A	A	C	
s	$i\backslash j$	0	1	2	3
A	0	0	−2	−4	−6
T	1	−2	2	0	2
A	2	−4	0	1	−1
G	3	−6	−2	[2]	0
	4	−8	−4	0	[1]

この場合も左上から求めたことが分かるので、u, v の先頭に $s[2], t[1]$ を加えて $u =$ “AG”, $v =$ “AC” として 2 行 1 列に移動する。

	t	A	A	C	
s	$i\backslash j$	0	1	2	3
A	0	0	−2	−4	−6
T	1	−2	2	0	2
A	2	−4	[0]	1	−1
G	3	−6	−2	[2]	0
	4	−8	−4	0	[1]

今度は上から求められた、つまり $A_{11}-2$ として求められたことが分かる。これは $s[1]$ にギャップが対応していることを意味するので、$u =$ “TAG”, $v =$ “-AC” として 1 行 1 列に移動する。

	t	A	A	C	
s	$i\backslash j$	0	1	2	3
A	0	0	−2	−4	−6
T	1	−2	[2]	0	2
A	2	−4	[0]	1	−1
G	3	−6	−2	[2]	0
	4	−8	−4	0	[1]

これは左上から求められたものなので、$u =$ “ATAG”, $v =$ “A-AC” として 0 行 0 列に移動し、終了する。このときの u, v が最良のアラインメントである。

	t	A	A	C	
s	$i\backslash j$	0	1	2	3
A	0	[0]	−2	−4	−6
T	1	−2	[2]	0	2
A	2	−4	[0]	1	−1
G	3	−6	−2	[2]	0
	4	−8	−4	0	[1]

このアルゴリズムを一般化すると、右下から i 行 j 列までのアラインメントが u, v に求められているときに、

1. A_{ij} と左、左上、上の値を比較する。
2. A_{ij} が左から求められた場合は、u の先頭にギャップを、v の先頭に $t[j-1]$ を追加し、左に移動する
3. A_{ij} が左上から求められた場合は、u, v の先頭に $s[i-1], t[j-1]$ をそれぞれ追加し、左上に移動する
4. A_{ij} が上から求められた場合は、u の先頭に $s[i-1]$ を、v の先頭にギャップを追加し、上に移動する

ことを 0 行 0 列に到達するまで続ける。最終的に得られた u, v が最良のアラインメントである。

7.5.3　動的計画法によるアラインメントのプログラム

動的計画法によってアラインメントを求めるプログラムは次の 5 つの関数からなる。

- ギャップ、一致、不一致の点数を定める g, q (ファイル 7.1 にあるものと同じなので省略)
- 表 A を作る関数 align (ファイル 7.2)
- トレースバックを行う関数 traceback (ファイル 7.3)
- 両者を使ってアラインメントを求める関数 align_dp (ファイル 7.4)

長さ m の文字列 s と長さ n の文字列 n をもとに表 A を作る関数 align(s,t) は、まず大きさ (m + 1)(n + 1) の配列を作り、0 行と 0 列を先に埋めておく。残りの部分は 12 行目からの繰り返しによって埋める。

```
1  def align(s,t)
2    m = s.length()
3    n = t.length()
4    a = make2d(m+1,n+1)
5    a[0][0] = 0
6    for j in 1..n
7      a[0][j] = a[0][j-1] + g()
8    end
9    for i in 1..m
10     a[i][0] = a[i-1][0] + g()
11   end
12   for i in 1..m
13     for j in 1..n
14       a[i][j] を計算する
15     end
16   end
17   a
18 end
```

ファイル 7.2: align.rb (関数 align)

練習 7.4 (アラインメント表の作成) ファイル 7.2 の 14 行目には、a[i][j-1], a[i-1][j-1], a[i-1][j] を用いて a[i][j] を求める命令が入る。ここを埋めて関数 align を完成させよ。(ヒント:ファイル 7.1 の 19 行目 から 22 行目)

```
1 irb(main):008:0> align("ATAG","AAC")
2 => [[0, -2, -4, -6], [-2, 2, 0, -2], [-4, 0, 1,
     -1], [-6, -2, 2, 0], [-8, -4, 0, 1]]
```

関数 traceback(a,s,t) はファイル 7.3 のようになる。引数 a は align によって作られた表 A である。この関数は表の右下から出発して、左上に到達するまでの繰り返しをするので、while 命令によって i, j が両方 0 になるまでの繰り返しを行っている。各 i, j の位置では左、左上、上のいずれかに移動するが、ここでは左に移る場合のみを示している (26 行目から 28 行目)。27 行目の式 t[j-1 .. j-1] は、文字列 t の (先頭を 0 文字目として)j-1 文字目を取り出す。

またこの関数は求めたアラインメント u, v の 2 つの文字列を答として返す必要がある。そのため、40 行目で大きさ 2 の配列を作って答としている。

最後に文字列 s, t のアラインメントを求める関数 align_dp(s,t) である。これは、align によって表を作り、traceback を使ってアラインメントを求

```
19 def traceback(a,s,t)
20   u = ""
21   v = ""
22   i = s.length()
23   j = t.length()
24   while i>0 || j>0
25     if j>0 && a[i][j] == a[i][j-1] + g()
26       u = "-"            + u
27       v = t[j-1 .. j-1] + v
28       j = j - 1 # go left
29     else
30       if i>0 && j>0 &&
31           a[i][j] == a[i-1][j-1] + q(s[i-1], t[j-1])
32         左上から求められた場合
33       else
34         if i>0 && a[i][j] == a[i-1][j] + g()
35           上から求められた場合
36         end
37       end
38     end
39   end
40   [u,v]
41 end
```

ファイル 7.3: `align.rb` (関数 `traceback`)

めるだけである。

```
42 def align_dp(s,t)
43   traceback(align(s,t),s,t)
44 end
```

ファイル 7.4: `align.rb` (関数 `align_dp`)

練習 7.5 (アラインメントを求めるプログラムの完成) ファイル 7.3 の 32 行目、35 行目を埋めてアラインメントを求めるプログラムを完成させよ。

```
1 irb(main):010:0> align_dp("ATAG","AAC")
2 => ["ATAG", "A-AC"]
3 irb(main):011:0> align_dp("GACG","GCAG")
```

```
4 => ["GAC-G", "G-CAG"]
```

練習 7.6 (RNA のアラインメント) 配布プログラム RNase_P.rb には 2 つの RNA 塩基配列を与える関数 seq0() および seq1() が定義されている。両者のアラインメントを align_dp を用いて求めよ。

練習 7.7 (スペルチェッカ) 辞書にある単語を並べた辞列 dict と、誤りのある単語 word が与えられたときに、word に最も似ている単語を答える spell(dict,word) を定義せよ。ただしここでの「最も似ている」とは、アラインメントの得点が最も高いものとする。

```
1 irb(main):018:0> spell(["align","airline","engine
    ","arrow"], "arigne")
2 => "align"
```

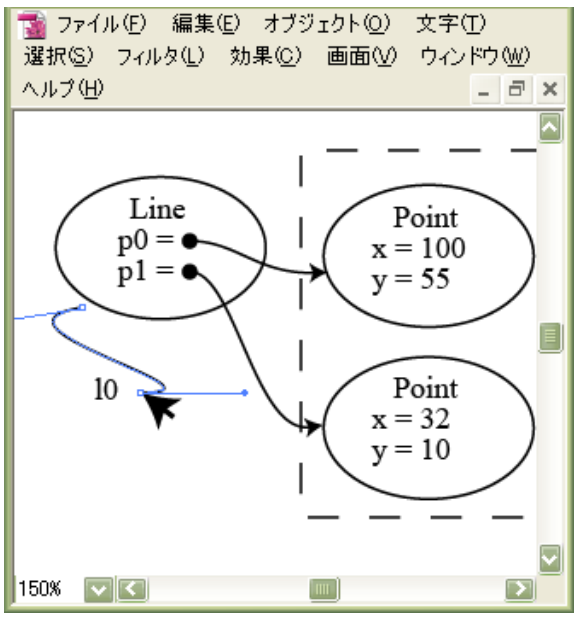
第8章　レコードとオブジェクト

これまでの章で見たプログラムが扱っていたデータは、数値、配列、文字列が中心だった。数学的な問題や文書の処理などであればこのようなデータだけでも充分な場合も多いだろう。しかし現実の情報システムは、より複雑なデータを扱わなければいけないことも多々ある。そのような場合には、**プログラム中でどのように複雑なデータを表わすか**が大きな問題となってくる。

この章では、複雑なデータを１つの値としてまとめた**レコード**とそれを発展させた**オブジェクト**を紹介する。オブジェクトはデータだけでなく、データの使い方までを１つの値にまとめたものであり、今日のソフトウェア開発でよく用いられている方法である。

8.1　複数の値をまとめるレコード

まずはどのような場合に複数の値をまとめたくなるのかを描画ソフトウェアを例に見てゆこう。描画ソフトウェアとは、マウスやペンなどを用いて線分、四角形、円、文字列などの図形要素を画面上に配置し、説明図や挿絵などを作画するものである。



このようなソフトウェアでは、線分や円などの 1 つ 1 つの図形について

- 位置
- 形状 (円であれば半径、四角形であれば幅や高さ)
- 色や模様

などの情報を記憶していて、それに基づいて表示や編集といった処理をする。画面に図を表示するときは、1 つ 1 つの図形が持っている情報を元に画面上に点を打ってゆき、利用者が図形を移動させる命令を出すと、その図形の位置情報を変更し、画面に図を表示し直すといった具合である。

このように 1 つの図形として見えるものには色々な種類の情報が含まれる。そのかたわら、「図形を削除する、コピーする」といった命令は図形を 1 つのデータとして扱う。このような場合、プログラムの中で**色々な種類の情報をまとめたデータ**を 1 つの値として扱うために、**レコード**というデータを作り操作する。

8.1.1　2 次元の点

簡単な例として、2 次元座標の点をレコードによって 1 つの値として表わそう。言うまでもなく 2 次元座標の点は、X 成分と Y 成分という 2 つの数値からなる。これを 1 つの値として表わすために、Ruby では**クラス**を定義する。

> 注意:　Ruby にはレコードを表わすための専用の手段は用意されていないので、本書では第 8.2 節以降で紹介する**オブジェクト指向プログラミング**のための機能を使ってレコードを表わしている。従って以降では、オブジェクト指向プログラミングの言葉を用いて説明をしている。

```
1 class Point
2   attr_accessor("x", "y")
3 end
```

ファイル 8.1: point.rb (一部)

この定義は「2 次元の点 (Point) には X 座標 (x) と Y 座標 (y) の情報が含まれる」ことを表わしている。この定義全体は**クラス定義**といい、x, y などの中に含まれる情報の名前を**インスタンス変数**という。

クラスが定義されると、Point.new() という式によって点を表わすデータ (**オブジェクト**という) を作ることができるようになる。例えば座標 (3,4) を表わす点を作るには、次のような命令を実行すればよい。

```
1  irb(main):004:0> load("./point.rb")
2  => true
3  irb(main):005:0> p = Point.new()
4  => #<Point:0x40332080>
5  irb(main):006:0> p.x = 3
6  => 3
7  irb(main):007:0> p.y = 4
8  => 4
9  irb(main):008:0> p
10 => #<Point:0x40332080 y=4, x=3>
```

3 行目は、Point クラスのオブジェクトを 1 つ作り、変数 p に代入している。続く行に表示されている#<Point:0x...>は作られたオブジェクトのクラス名と、値がしまわれているメモリ上の番地を表わしている。この時点でオブジェクトの中身は空である。

5 行目と 7 行目は X 座標として 3 を、Y 座標として 4 を代入している。ここで用いられている p.x という書き方は、変数 p にしまわれているオブジェクトの中の x というインスタンス変数を表わす。変数と同じように代入や参照をすることができるが、しまわれる場所がオブジェクト の中である点が異なる。9 行目のように p にしまわれているオブジェクトをもう一度表示すると、今度はオブジェクトの中に x や y の値がしまわれていることが分かる。

オブジェクトの中にしまわれているインスタンス変数を参照する場合には、次のような式を書く。

```
1  irb(main):013:0> p.y
2  => 4
3  irb(main):014:0> sqrt(p.x ** 2 + p.y ** 2)
4  => 5.0
```

これらの式は、p にしまわれているオブジェクトの中にあるインスタンス変数 x, y を参照している。

以下では、この 2 次元の点を用いて座標を計算して作図をする。そこで、作図の際に必要とな関数を定義しておこう (ファイル 8.2)。

点を作る: 座標 (u, v) を表わす点を作るには、すでに見たように

p = Point.new() ↵ p.x = u ↵ p.y = v

という 3 つの命令を実行すればよい。ただし、この操作は何度も行われるので、1 つの関数 point_make(u,v) としてまとめてしまおう。ファイル 8.2 の 4 行目からの定義がそれである。考え方は、0 で埋められた配列を作る関数 (p.38, 練習 3.2) などと同じである。

```
4  def point_make(u,v)
5    p = Point.new()
6    p.x = u
7    p.y = v
8    p
9  end
10
11 def point_scale(p,s)
12   point_make(p.x*s,p.y*s)
13 end
14
15 def point_add(p,q)
16   point_make(p.x+q.x, p.y+q.y)
17 end
18
19 def point_interpolate(p,q,t)
20   point_add(point_scale(p,1-t),
21             point_scale(q,t))
22 end
23
24 def point_draw(p,a)
25   if 0 <= p.y+0.5 && p.y+0.5 < a.   length() &&
26      0 <= p.x+0.5 && p.x+0.5 < a[0].length()
27     a[p.y+0.5][p.x+0.5]=1
28   end
29 end
```

ファイル 8.2: point.rb (続き)

スカラー倍: 座標 (x, y) を表わす点 p を s 倍した座標 (sx, sy) を表わす点を作る関数 point_scale(p,s) は 11 行目から定義されている。この関数は**新しく点を作る**。つまり p が $(3, 4)$ を表わしているときに q = point_scale(p,2) という命令を実行しても p の位置は変わらず、新しく作られた $(6, 8)$ を表わす点が q に代入される。

ベクトル和: 座標 (x, y) を表わす点 p と座標 (u, v) を表わす点 q のベクトル和である座標 $(x + u, y + v)$ を表わす点を作る関数は point_add(p,q) は 15 行目から定義されている。これも新しい点を作っていることに注意しよう。

線形補間: 2 つの点 p, q を t : $(1-t)$ に内分する点を求める関数 point_interpolate(p,q,t) は point_scale, point_add を組み合わせることで定義されている (19 行目から)。この内分点はベクトル演算では $(1-t)\mathrm{p}+t\mathrm{q}$ という式で求めることができるので、point_scale と point_add の組み合わせによって定義できる。

点を打つ: 画像を表わす 2 次元配列 a の点 p の位置に点を打つ関数 point_draw(p,a) は 24 行目から定義されている。ただし、点 p の座標に数値誤差がある場合を考えて各座標には 0.5 を足すことで四捨五入した位置に点を打つ。また p は配列の中に収まるとは限らないので、座標が配列の大きさに収まる場合にのみ点を打つ。

8.1.2　直線

レコードを使うことによって、複雑とまでははは言えないが、2 つの座標データを持つ 2 次元の点を 1 つのデータにまとめることができた。この効果は、点を使った様々な処理が単純に分かりやすく表わせることで実感できる。

そこで 2 次元の画像に直線と曲線を描く方法を見てゆこう。画像は第 2.1 節で紹介したように 2 次元配列を使って表わし、isrb 中の関数 show によって表示することにする。

まずは 2 点 $p_0=(x_0,y_0)$, $p_1=(x_1,y_1)$ の間の線分を作図する関数である。

```
1 load("./max.rb")
2 load("./abs.rb")
3
4 def line_draw(p0,p1,a)
5   n=max(abs(p1.x - p0.x), abs(p1.y - p0.y))
6   for i in 0..n
7     point_draw(point_interpolate(p0,p1,i*1.0/n), a)
8   end
9 end
```

ファイル 8.3: line.rb

定義を見る前に線分の描き方を考えておこう。まずこの線分は

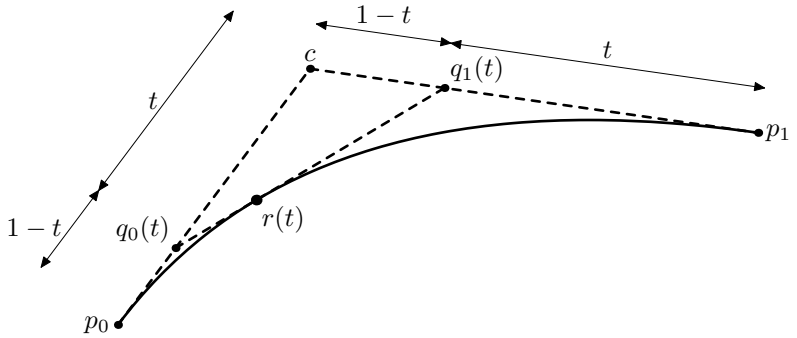
$$p(t)=(1-t)p_0+tp_1 \quad (0\le t\le 1) \tag{8.1}$$

と書くことができる。従って t の値を 0 から 1 まで Δt ずつ変化させて $p(t)$ に対応する位置に点を打てばよい。Δt は、p_0,p_1 の X,Y 座標ごとの差の絶対値の大きい方を n としたときに $\Delta t=1/n$ とすれば、$p(t)$ と $p(t+\Delta t)$ は水平方向あるいは垂直方向に 1 離れるので、点が途切れることなくつながる。

この考え方を関数としたものが line_draw である。$p(t)$ を求めるところと点を打つところは、すでに定義した point_interpolate と point_draw が使えるので、式 8.1 をほとんどそのまま関数として定義できている。これがレコードを使う効果である。

8.1.3　曲線

曲線を描く方法には色々なものがあるが、ここでは、3 点 p_0, c, p_1 が与えられたときに次の図のような曲線を描くことにする。この曲線は 2 次の Bézier 曲線と呼ばれている。



この曲線は以下の式によって定められる $r(t)$ を動かしたときの軌跡である。

$$\begin{aligned} q_0(t) &= (1-t)p_0 + tc \\ q_1(t) &= (1-t)c + tp_1 \qquad (0 \le t \le 1) \\ r(t) &= (1-t)q_0(t) + tq_1(t) \end{aligned} \tag{8.2}$$

つまり、p_0, c と c, p_1 という 2 点を $(1-t) : t$ に内分する点をそれぞれ求め、その内分点どうしの内分点が $r(t)$ である。

ファイル 8.4 に示すのが 3 点 p0, c, p1 によって定められる Bézier 曲線を 2 次元配列 a 上に描く関数 bezier_draw(p0,c,p1,a) である。この関数は曲線を n 本の線分で近似して作図している。i 番目の線分を描くときはまず $r(\mathtt{i}/\mathtt{n})$ の座標を求め、prev にしまわれている $\mathtt{i}-1$ 番目の点との間に線分を引いている。その後、prev に現在の点を代入することで次回の繰り返しにおける prev が $\mathtt{i}-1$ 番目の点になるようにしている。

練習 8.1 (連続する曲線による図) 点の列が与えられたときに、順に曲線でつないでゆくと、一筆描きの図を簡単に描くことができる。ファイル 8.5 に示した関数 kana() は、配列として与えられた XY 座標から点の配列 p を作り、p の $2i, 2i+1, 2i+2$ 番目の 3 点による Bézier 曲線を描くものである。

a) この関数を isrb の中で実行して、どのような図形が表示されるかを確認せよ。

```
1  load("./line.rb")
2  def bezier_draw(p0,c,p1,a)
3    n = 10
4    prev = p0
5    for i in 1..n
6      t = i*1.0/n
7      q0 = point_interpolate(p0, c,  t)
8      q1 = point_interpolate(c,  p1, t)
9      r  = point_interpolate(q0, q1, t)
10     line_draw(prev, r, a)
11     prev = r
12   end
13 end
14
```

ファイル 8.4: bezier.rb

b) この関数を参考にして、自分でデザインした図形を描く関数mypicture()を定義せよ。

なお、5 行目からの make_points(xy) は、X 座標と Y 座標が交互に並んだ配列から、Point オブジェクトの配列を作る関数である。

練習 8.2 (3 次曲線) 式 8.2 を展開すると $r(t)$ は t の 2 次関数であることが分かる。つまりこの式によって描かれるのは 2 次曲線 (放物線) である。ところで 3 次の Bézier 曲線は、4 点 p_0, c_0, c_1, p_1 が与えられたときに以下のような式によって定められる $s(t)$ の軌跡である。

$$q_0(t) = (1-t)p_0 + tc_0, \quad q_1(t) = (1-t)c_0 + tc_1, \quad q_2(t) = (1-t)c_1 + tp_1$$
$$r_0(t) = (1-t)q_0 + tq_1, \quad r_1(t) = (1-t)q_1 + tq_2$$
$$s(t) = (1-t)r_0 + tr_1$$

a) この式をもとに、3 次の Bézier 曲線を描く関数bezier_draw3(p0,c0,c1,p1,a)を定義せよ。

b) 次の点列を使って一続きの曲線を描く関数 kana3() を定義せよ。

$$(323, 25), (210, 553), (-1, 361), (0, 242), (-1, -71), (242, -24),$$
$$(269, 234), (307, 448), (390, 347), (399, 302)$$

(練習 8.1 の図形は 5 本の曲線から成っていたのに対して、この図形は 3 本の曲線である。) ただし先頭を 0 番目として、$3i, 3i+1, 3i+2, 3i+3$ 番目の点を使って 1 つの Bézier 曲線を描くものとする。

```
load("./point.rb")
load("./bezier.rb")
load("./make2d.rb")

def make_points(xy)
  points = Array.new(xy.length()/2)
  for i in 0 .. points.length()-1
    points[i] = point_make(xy[2*i], xy[2*i+1])
  end
  points
end

def kana()
  a=make2d(400,400)
  p=make_points([208,70, 163,210, 56,317, 0,264,
                 0,196, 115,50,  250,53, 353,67,
               390,149, 412,268, 237,347])
  for i in 0..(p.length())/2-1
    bezier_draw(p[i*2], p[i*2+1], p[i*2+2], a)
  end
  show(a)
end
```

ファイル 8.5: kana.rb

c) この関数を参考にして、自分でデザインした図形を描く関数mypicture3()を定義せよ。

8.1.4　レコードを使う意義

この章の最初で、レコードは複雑なデータを 1 つの値としてまとめるものだと説明した。関数 bezier_draw(ファイル 8.4) の定義に即して考えてみると、レコードを使うことには次のような意義があることが分かる。

1 つの点を表わすために変数が 1 つで済む。 レコードを使わない場合、X 座標、Y 座標をしまう別々の変数が必要であるが、bezier_draw では p0 や q0 のように 1 つの変数で 1 つの点を表わすことができている。

計算した点を答えとするような関数を定義できる。 例えば内分点を求める point_interpolate(p0,p1,t) はレコードによって表わされた 1 つの

点を答える関数であった。レコードを使わない場合、1 つの関数は 1 つの値を答えることしかできないので、X 座標と Y 座標についてそれぞれ内分する関数を使うことになるだろう。

ここでは簡単のために 2 次元の点を使って説明していたため X 座標と Y 座標の 2 つのデータを 1 つのデータにできた効果しか見られなかった。そのため一見、大きな違いはないと思われるかも知れない。しかし、より複雑なデータを表現する場合には XY 座標だけなく沢山の種類のデータを 1 つにまとめることになるため、レコードを使う意義が大きくなる。

練習 8.3 (レコードを使わない曲線) $p_0 = (\mathtt{x0}, \mathtt{y0}), c = (\mathtt{xc}, \mathtt{yc}), p_1 = (\mathtt{x1}, \mathtt{y1})$ によって定義される 2 次の Bézier 曲線を描く関数 `bezier_nr(a,x0,y0,xc,yc,x1,y1)` をレコードを使わずに定義せよ。ファイル 8.4 と比べて定義の分かりやすさは違うだろうか? (注意: 大きさ 2 の配列に X 座標と Y 座標をしまうことで 1 つの点を表わす方法もあるが、このようなやり方もしないものとする。これは配列によってレコードを表わしていることに相当する。)

練習 8.4 (円) 次の手順に従って円を描く関数を定義せよ。

a) 点 `p` を原点中心に角度 `theta` だけ回転させた座標を求める関数 `rotate(p,theta)` を定義せよ。ただし座標 (x, y) を原点まわりに θ だけ回転した座標は以下のように求められる。

$$(x\cos\theta - ysin\theta, x\sin\theta + y\cos\theta)$$

b) 点 `p` を中心に半径 `r` の円を、2 次元配列 `a` に描くような関数 `circle_draw(p,r,a)` を定義せよ。(ヒント: 曲線を描くときと同じように、何本かの直線で近似すればよい。例えば、N 本の直線で近似させるのであれば、i 番目の点は点 $(r, 0)$ を原点中心に $2\pi i/N$ だけ回転させ、それに `p` を加えることで決められる。)

8.2　値と操作をまとめるオブジェクト指向

現実の情報システムは沢山の種類の値を扱わなければいけないことが多い。

一方、レコードを使って複雑な値を表わす場合、その値に対する計算や操作を同時に定義しなければいけないことはすでに見た。例えば、2 次元の点であれば、点を作る、スカラー倍、ベクトル和などである。

値の種類が増えると、それぞれの値について計算や操作を定義することになる。例えば、直線、曲線、円などの図形であれば次のようなクラスや関数を定義することになるだろう。

直線クラス	曲線クラス	円クラス
直線を作る	曲線を作る	円を作る
直線を表示する	曲線を表示する	円を表示する
直線を回転する	曲線を回転する	円を回転する
直線を拡大縮小する	曲線を拡大縮小する	円を拡大縮小する
直線を平行移動する	曲線を平行移動する	円を平行移動する
	⋮	

そしてこれらの値が混在しているときには、値の種類に応じて関数を使い分けなければいけない。例えば、直線、曲線、円をそれぞれレコードを使って表わし、それらが混在する配列 f を表示する関数 drawall(f,a) を定義することを考える。図形の種類によって表示させる方法が違うので、直線であれば line_draw 曲線であれば bezier_draw を使うように定義しなければならないので、関数の定義は次のようになるだろう。

```
1  def drawall(f,a)
2    for i in 0..f.length()
3      if f[i]が直線の場合
4        line_draw(f[i]の両端点,a)
5      else
6        if f[i]が曲線の場合
7          bezier_draw(f[i]の端点と制御点,a)
8        else
9          if f[i]が円の場合
10           circle_draw(f[i]の中心と半径,a)
11         end
12       end
13     end
14   end
15 end
```

また例えば「図形を点 (x, y) を中心に角度 θ 回転させる」場合には図形の種類に関らず

図形を $(-x, -y)$ だけ平行移動して
原点中心に角度 θ 回転させ
図形を (x, y) だけ平行移動する

という操作をすればよいように思える。しかし、平行移動や回転をさせる関数は図形の種類ごとに定義されているので、drawall の場合と同様に、図形の種類で場合分けをしなければならない。

オブジェクト指向は、データに対する操作をデータに覚えさせるようなプログラムの作り方である。図形の編集で言えば、「表示」「移動」などの操作方法は各図形が覚えており、図形を扱う側は図形の種類による違いを意識する必要がなくなる。

オブジェクト指向はまた、プログラムを再利用するのに適している。ここで言う再利用とは、似た種類のデータ (オブジェクト) に対する操作をまとめて定義してしまうような方法である。例えば直線と曲線は多くの性質や操作が共通しているので、それらをまとめることができる。

8.2.1　点を表わすオブジェクト

では第 8.1 節でレコードとして表わされていた 2 次元の点をオブジェクトとして定義し直してみよう。定義全体はファイル 8.6 のような Point という名前のクラスの定義になる。レコードのときと同じくインスタンス変数 (2 行目の attr_accessor) の定義があるが、それに加えて Point クラスのオブジェクトに対する計算や操作 (4 行目以降) をまとめたものになっている。

```
1  class Point
2    attr_accessor("x", "y")
3
4    def initialize(u,v)
5      self.x = u
6      self.y = v
7    end
8
9    def scale(s)
10     Point.new(self.x * s, self.y * s)
11   end
12
13   def add(q)
14     Point.new(self.x + q.x, self.y + q.y)
15   end
16 end
```

ファイル 8.6: oo-point.rb

4 行目以降にある関数のような定義は、クラス定義の中に書かれているために、これまでの関数と少しだけ違った動きをする。本書では、このようなクラス定義中にあるものを**メソッド**と呼び、これまでのようにクラス定義の外で定義された**関数**と区別する。

4 行目から定義されている initialize というメソッドは、**初期化メソッド**とよばれるもので、オブジェクトが作られた際に自動的に呼び出されるものである。この定義があると、下に示すように Point.new(3,4) という式だけでインスタンス変数 x, y に 3,4 が代入された Point オブジェクトを作ることができる。

```
1 irb(main):005:0> p = Point.new(3,4)
2 => #<Point:0x4032d544 y=4, x=3>
3 irb(main):006:0> p.y
4 => 4
```

この初期化メソッドの中では self.x に代入しているが、self とは**現在注目しているオブジェクト自身**を表わす特別な変数である。つまり Point.new(3,4) を計算するときは

1. Point クラスのオブジェクトが作られる
2. 作られたオブジェクトの初期化メソッドが引数 3,4 とともに実行される
3. 初期化メソッドの中で self は、ここで作られたオブジェクト を表わしているので、そのオブジェクトのインスタンス変数 x, y に 3, 4 が代入される

という操作が行われることになる。前節のファイル 8.2 に定義された関数 point_make の定義と見比べて欲しい。前節の定義が「点 p を作るときに p のインスタンス変数 x, y に代入する」ものだったのに対し、オブジェクト指向では「わたしが作られるときは、わたし (self) のインスタンス変数 x, y に代入する」ものになっている。このようにオブジェクト指向は**オブジェクトの視点から定義をする**方法なのである。

9 行目から定義されているメソッド scale(s) は、ある点を s だけスカラー倍した点を作る。このメソッドを使う書くには、例えば p が Point オブジェクトだったら、p.scale(2) という式になる。

配列 a の長さを求める式 a.length() もこの書き方と同じである。

```
1 irb(main):008:0> q = p.scale(2)
2 => #<Point:0x4031ed64 y=8, x=6>
```

メソッド scale の定義と前節の point_scale(p,s)(ファイル 8.2 の 11 行目) を見比べてほしい。前節では「p を s 倍した点を新しく作る」定義だったものが、オブジェクト指向では「わたし (self) を s 倍した点を新しく作る」定義になっている。結果としては p を self に置き換えて、点を作るために Point.new(x,y) という式を使うようになっただけだということが分かるだろう。

2 つ点のベクトル和を求める add メソッドは 13 行目のような定義になる。Ruby のメソッドは 1 つのオブジェクトに対するものなので、このような 2 つ

のオブジェクトを用いた計算は、1 つ目のオブジェクトのメソッドの引数に 2 つ目のオブジェクトを与えるような書き方になる。結果として点 p, q のベクトル和は p.add(q) のように p に対してメソッドを呼び出すことになる。

.add を + 記号に置き換えれば「$p+q$」と読めるだろう。

```
1 irb(main):010:0> p.add(q)
2 => #<Point:0x4031b4c0 y=12, x=9>
3 irb(main):011:0> p.add(q).scale(0.5)
4 => #<Point:0x4031636c y=6.0, x=4.5>
```

練習 8.5 (点オブジェクトのメソッド) Point クラスの定義 (ファイル 8.6) に、以下の操作を行うメソッド定義を追加せよ。

a) 点 q との差を求める sub(q)

b) 2 次配列 a 上に点を打つ draw(a)

c) 点 q と $(1-t):t$ の内分点を求める interpolate(q,t)

d) 原点中心に角度 theta だけ回転した位置を求める rotate(theta)

ただし座標 (x, y) を原点まわりに θ だけ回転した座標は以下のように求められる。

$$(x\cos\theta - ysin\theta, x\sin\theta + y\cos\theta)$$

8.2.2　図形を表わすオブジェクト

続けて直線や曲線といった図形を表わすオブジェクトをいくつか定義してみよう。まず直線オブジェクトの定義はファイル 8.7 のようになる。

ここでは簡単のために初期化メソッドのほかは次のメソッドだけが定義されている。

- draw(a) というメソッドを呼び出すと、2 次元配列 a に直線を描く
- turn(theta) というメソッドを呼び出すと、原点中心に角度 theta だけ回転した位置へ移動する

初期化メソッドと draw の定義自体は前節の line_make と line_draw をオブジェクト指向のやり方で書き直しただけなので説明は不要だろう。

メソッド turn は、新しく直線を作るのではなく、呼び出された直線オブジェクトの状態を変える。Point オブジェクトに定義されている rotate(theta) メソッドは、角度 theta だけ回転した座標の点を新しく作るので、21 行目や次の行のように p0, p1 に新しく計算した点を代入している。

練習 8.6 (円クラス) 円を表わすクラス Circle を定義し、次のようなメソッドを定義せよ。

```
load("./max.rb")
load("./abs.rb")
class Line
  attr_accessor("p0", "p1")

  def initialize(q,r)
    self.p0 = q
    self.p1 = r
  end

  def draw(a)
    n = max(abs(self.p1.x - self.p0.x),
            abs(self.p1.y - self.p0.y))
    for i in 0..n
      p = self.p0.interpolate(self.p1, i*1.0/n)
      p.draw(a)
    end
  end

  def turn(theta)
    self.p0 = self.p0.rotate(theta)
    self.p1 = self.p1.rotate(theta)
  end
end
```

ファイル 8.7: oo-line.rb

- 中心を p に、半径を r とする初期化メソッド initialize(p,r)
- 2 次配列 a に円を描く draw(a)
- 原点中心に theta 回転した位置に移動する turn

8.2.3　継承によって図形の種類を追加する

曲線は直線とよく似た性質を持っており、(2 次の Bézier 曲線であれば) 直線に制御点を 1 つ追加しただけのものと見なせる。このような場合、曲線を表わす Bezier クラスを定義する際に、Line のクラス定義を**継承**することで、重複する定義を省略することができる。

それではクラス Bezier の定義 (ファイル 8.8) を見てゆこう。

```
1  load("./oo-line.rb")
2  class Bezier < Line
3    attr_accessor("p0", "c", "p1")
4
5    def initialize(q,r,s)
6      super(q,s)
7      self.c  = r
8    end
9
10   def draw(a)
11     n = 10
12     prev = self.p0
13     for i in 1..n
14       t = i*1.0/n
15       q0 = self.p0.interpolate(c,  t)
16       q1 = self.c. interpolate(p1, t)
17       r  = q0.interpolate(q1, t)
18       Line.new(prev, r).draw(a)
19       prev = r
20     end
21   end
22
23   def turn(theta)
24     super(theta)
25     self.c = self.c.rotate(theta)
26   end
27 end
```

ファイル 8.8: oo-bezier.rb

クラスの継承　クラス定義の先頭 (2 行目) には、クラス名 Bezier に続けて< Line と書かれている。これは、ここで定義しているクラスが Line クラスの定義を継承することを意味している。この指示によって Bezier クラスは Line クラスが持っていた全てのメソッド定義をはじめから持っていることになる。

このとき Bezier クラスを**子クラス**、Line クラスを**親クラス**と呼ぶ。

メソッドの継承と上書き　Line クラスにあった initialize, draw, turn などのメソッドの定義は Bezier クラスにそのまま継承されるので、Bezier ク

ラスの中に何も定義しない場合は、Bezier クラスは Line クラスと全く同じ働きをする。

そこで draw メソッドによって曲線を表示するために、Bezier クラス独自のメソッドを 10 行目 から定義する。このように、親クラスにあるメソッドと同じ名前でメソッドを定義することを**上書き**という。

Bezier オブジェクトに対して draw メソッドを呼び出すと上書きされた方のメソッドが実行される。これによって「他は Line と同じだけど表示方法だけが違う」定義が可能になる。

ここでは初期化メソッド (initialize) とメソッド turn も同様に上書きしている。

親クラスのメソッドの実行　23 行目からの turn メソッドに注目してほしい。Line クラスの turn メソッドは 2 つの端点の位置を原点まわりに回転させるものだった。Bézier 曲線は 2 つの端点に加えて制御点があるので、回転をする場合には制御点も一緒に移動させなければいけない。

このような場合、まず Line クラスの turn を使うことで両端点を回転させ、Bezier クラスに追加された制御点を回転させることで全体を回転させることができる。ここでは 24 行目に書かれた命令 super(theta) は、親クラスに定義されている同じ名前のメソッド (turn) を実行させている。

練習 8.7 (点のまわりの回転) 点 p のまわりに図形を theta だけ回転させるメソッド turn_at(p,theta) を定義したい。点 p のまわりで回転させるには (1) 図形を −p だけ平行移動させ (2) 原点中心に回転させ、(3)p だけ平行移動すればよい。もし move(p) というメソッドで図形を p だけ平行移動できたとすると、次のようなメソッドが定義されていればよいことになる。

```
1 def turn_at(p,theta)
2   self.move(p.scale(-1)) #-pだけ平行移動
3   self.turn(theta)       #原点中心に回転
4   self.move(p)           #pだけ平行移動
5 end
```

この定義を Line, Bezier および練習 8.6 で定義した Circle の 3 つのクラスに共通して使いたいので、次のように定義せよ。

a) 図形を点 p だけ平行移動させるメソッド move(p) を Line, Bezier および練習 8.6 で定義した Circle の 3 つのクラスにそれぞれ定義せよ。

b) Figure というクラスを定義してその中に上に示したメソッド turn_at の定義を書け。

c) クラス Line と Circle が Figure の子クラスとなるようにそれぞれのクラス定義を変更せよ。これで例えばファイル 8.9 の turning_figure

の e ように 1 つのメソッド定義で異なる種類の図形を回転させることができるようになる。

```
include(Math)
def turning_figure()
  f1 = Line.new(Point.new(0,0),Point.new(99,99))
  f2 = Circle.new(Point.new(50,50),25)
  a = make2d(100,100)
  for i in 0..10
    f1.draw(a)
    f2.draw(a)
    f1.turn_at(Point.new(100,100),  PI/40)
    f2.turn_at(Point.new(100,100), -PI/40)
  end
  show(a)
end
```

ファイル 8.9: turn_at.rb

練習 8.8 (3 次曲線のクラス) 3 次の Bézier 曲線を表わすクラス Bezier3 を、Bezier を継承して定義せよ。(3 次の Bézier 曲線の描き方は練習 8.2 を参照せよ。)

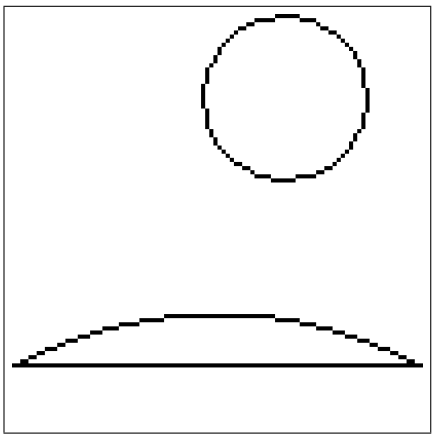
8.2.4　多相性によって異なる種類の図形を統一的に扱う

オブジェクトがどのような計算・操作されるべきかを自分で覚えていることで、色々な種類 (クラス) のオブジェクトを統一的に扱うことが可能になる。このことを**多相性**(polymorphism) という。

ファイル 8.10 に示した関数 drawmoon を見てほしい。この関数は、直線 (Line)、曲線 (Bezier)、円 (Circle) の 3 種類のオブジェクトが入った配列 f を作り、次に示す関数 drawall(ファイル 8.11) を使って表示している。

```
1  def drawmoon()
2    p0=Point.new(0,85)
3    p1=Point.new(99,85)
4    f=[Line.new(p0,p1),
5       Bezier.new(p0,Point.new(50,60),p1),
6       Circle.new(Point.new(66,20),20)]
7    a=make2d(100,100)
8    drawall(f,a)
9    show(a)
10 end
```

ファイル 8.10: drawall.rb (一部)



```
11 def drawall(elements,a)
12   for i in 0..elements.length()-1
13     elements[i].draw(a)
14   end
15   a
16 end
```

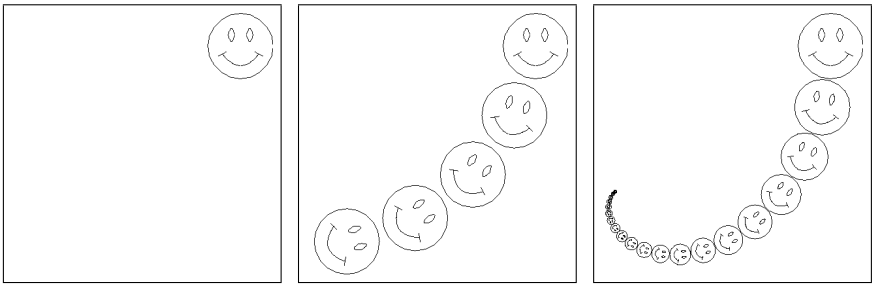
ファイル 8.11: drawall.rb (続き)

関数 drawall(f,a) の定義を見ると、配列の内容 1 つ 1 つについて draw メソッドを呼び出している (13 行目) だけである。つまり、配列の中にある値が直線であれ何であれ、そのオブジェクト の draw メソッドを使えば適切に表示されることになっているので、drawall を定義するときはこれで上手くゆく。これが多相性によって可能になったことである。

練習 8.9 (作図) ファイル 8.10 や配布プログラム `oo-face.rb`(後述) を参考に、適当な絵を表わす配列を返す関数 `sketch` を定義せよ。正しく定義されたものは `drawall` を使って表示させることができるはずである。

練習 8.10 (図形の回転) 配列 `f` にしまわれた全ての図形をそれぞれ原点のまわりに `theta` だけ回転させる `turnall(f,theta)` を定義せよ。

練習 8.11 (図形の変形) 配布プログラム `oo-face.rb` に定義されている関数 `face` は、円、曲線、直線がしまわれた配列を作る。その配列を `drawall` を使って 400×400 の画像に表示すると以下の左の図のようになる。



関数 `face` によって作られる図形を変形して、上の中央と右のような図を以下の手順に従って作成せよ。

a) 図形を原点を中心として `s` 倍に拡大・縮小させるメソッド `zoom(s)` ををクラス `Line`, `Circle`, `Bezier` に定義せよ。ただし拡大・縮小の中心は原点だとする。

b) ファイル 8.12 に定義された関数 `faces`, `whirl` を使うとそれぞれ上の中央と右の図が表示されるはずである。確めよ。

c) ここに示したものを参考に、回転・移転・拡大縮小を使って面白いパターンを描く関数 `art()` を定義せよ。

8.3　定義のまとめ

クラスの定義: クラス定義は次のような形をしている。

```
class 名前1 < 名前2
  attr_accessor("変数名1","変数名2",...)
  メソッド定義1
  メソッド定義2
  ⋮
end
```

```
include(Math)
def faces()
  a = make2d(400,400)
  elements = face()
  for i in 1..5
    drawall(elements, a)
    turnall(elements, PI/10)
  end
  show(a)
end

def whirl()
  a = make2d(400,400)
  elements = face()
  for i in 1..20
    for j in 0..elements.length()-1
      elements[j].draw(a)
      elements[j].turn(PI/15)
      elements[j].zoom(0.85)
      elements[j].move(Point.new(55,40))
    end
  end
  show(a)
end
```

ファイル 8.12: oo-whirl.rb

名前$_1$ は新しく定義されるクラスの名前である。名前$_2$ はすでであるクラスの名前で**親クラス**と呼ばれる。「< 名前$_2$」の部分は省略することができる。

インスタンス変数: 1 つ 1 つのオブジェクトの中にある変数をインスタンス変数 という。クラス定義に

attr_accessor("変数名$_1$","変数名$_2$",...)

と書くとそのクラスのオブジェクトは変数名$_i$ という名前のインスタンス変数 を持つようになる。attr_accessor に書く変数名は " " で囲まなければいけないことに注意せよ。

初期化メソッド: あるクラスのオブジェクトが作られるときに呼び出されるメソッドを初期化メソッドという。Ruby では `initialize` という名前のメソッドを定義すると、それが初期化メソッドとなる。

自分自身を表わす変数 `self`: メソッド定義の中では、現在そのメソッドを実行しているオブジェクト自身を `self` という名前の変数で表わす。

親クラスのメソッドを呼び出す `super`: メソッド中で `super(`式$_1$`,`式$_2$`,...)` という式が計算されると、現在実行中のメソッドと同じ名前の親クラスに定義されたメソッドが実行される。

オブジェクトの作成: あるクラスのオブジェクトを作る式は

クラス名`.new(`式$_1$`,`式$_2$`,...)`

という形をしている。この式が計算されると、クラス名のオブジェクト が 1 つでき、そのオブジェクトの `initialize` メソッドが自動的に実行される。

インスタンス変数の読み書き: 式`.`変数名という式は、式が表わすオブジェクトの変数名というインスタンス変数を参照する。

式$_1$`.`変数名`=`式$_2$という命令は、式$_1$が表わすオブジェクトの変数名というインスタンス変数に式$_2$の値を代入する。

メソッドの実行: 式$_0$`.`メソッド名`(`式$_1$`,`式$_2$`,...)` という式は、式$_0$が表わすオブジェクトのメソッドを呼び出す。式$_0$が表わすオブジェクトのクラスにメソッド名を持つメソッドが定義されている場合は、そのメソッドが実行される。定義されていない場合は、親クラスを順に辿ってゆき最初に見つけたメソッド名を持つメソッド を実行する。

第９章　データ構造と再帰

準備中

第10章　いろいろなプログラミング言語

この章はプログラムを書くための言語—プログラミング言語—について概観する。本書では**Ruby 言語**を用いて情報科学の概念や方法を解説してきたが、実際の情報システムの開発や情報科学の研究開発においては、Ruby 以外にも数多くのプログラミング言語が用いられている。

プログラミング言語には様々な特徴を持ったものがある。例えば数学や人工知能の問題を解くといった特定の目的のプログラムを簡単に作れるように工夫されているものや、プログラムの実行速度ができるだけ速くなるように設計されたもの、プログラムを作るための手間が少なくなるように工夫されたものなど様々である。Ruby はそのように沢山あるプログラミング言語の一つに過ぎない。

そこで、この章ではまずプログラミング言語がどのように発展してきたかを概観して、いくつかの代表的なプログラミング言語がどのような特徴を持っているかを紹介する。最後に、Ruby 以外の言語を使って実際にプログラムを動かしてみて、実行方法や、速度の違いを見ておこう。

10.1　プログラミング言語の発展

プログラミング言語とは、言うまでもなくコンピュータが実行するプログラムを書くために作られた「言語」である。

「情報」(川合編, 東京大学出版会) どこ?参照。

10.1.1　機械語とアセンブリ言語

コンピュータのハードウェアが実行しているのは、四則演算のような単純な命令を並べて作られた**機械語プログラム**と呼ばれるものである。コンピュータが発明された当初は、人間が機械語プログラムを直接作っていた。例えば $(D + C)(D - C)$ という計算をして A にしまうのであれば、これを足し算や掛け算の命令に分解して次のような命令の並びを作ることになる。

命令	アセンブリ言語による表現	メモリ上の表現
D の値を A にしまう	`movl %edx, %eax`	9, 80
A を C だけ減らす	`subl %ecx, %eax`	41, 72
C を D だけ増やす	`addl %edx, %ecx`	1, 81
A を C 倍する	`imull %ecx, %eax`	15, 47, 65

実際には、右端の列にあるような $9, 80, 41, \ldots$ 数値がメモリ上に並べられ、それをハードウェアが読み取って計算を行うことになる。人間がプログラムを書く場合には、中央の列にあるような単語を並べて書き表わす。この書き方は**アセンブリ言語**と呼ばれ、1 つ 1 つの行が右列の数値として表現された命令に対応している。

上の例を見て分かるように、機械語のプログラムは沢山の命令を並べて 1 つの式や操作を表わすことになるため、人間がこれを直接書き表わすのは非常に労力のいる作業である。

さらに、機械語プログラムはハードウェアの間での**互換性**がないこともある。つまりハードウェア、正確には CPU, によって全く違った機械語命令を用意しており、それらに割り当てている数値も違うため、ある種類の CPU のために作られた機械語プログラムは、別の種類の CPU では実行できないのが普通である。逆に言えば、機械語でプログラムを作っていると、新しいハードウェア (CPU) のためにはプログラムを全て作り直さなければいけなかった。

10.1.2　高級プログラミング言語の誕生と発展

その後、より人間に分かりやすい書き方でプログラムを作ることができるようになった。これは**高級言語**と呼ばれるもので、我々が学んだ Ruby もそのひとつである。現在では単にプログラミング言語と言った場合には高級言語のことを意味することが多い。

機械語に対して「高級」だという意味である。

最も初期の (高級) プログラミング言語の代表は 1950 年代に発明された **FORTRAN**, **COOBL**, **LISP** などである。FORTRAN は科学技術計算、COBOL は事務処理、LISP は人工知能研究などの記号処理を目的に設計された。これらの言語は今日でも使われている。また、その後に作られた多くのプログラミング言語の基礎にもなっている。

1960 年代から 1970 年代にかけて新しい考え方をとり入れた言語が数多く作られた。代表的なものには**オブジェクト指向**の考えをとり入れた Simula や Smalltalk, 初学者向けに設計された BASIC や Pascal, オペレーティングシステムのようなハードウェアを制御する目的で設計された C, **論理型**言語 Prolog, **関数型**言語 ML などがある。

1980 年代から 1990 年に入って、これらの言語を様々目的のために発展させたり機能を追加した言語が作られた。代表的なものとして、C にオブジェクト指向の考え方をとり入れた C++, 文字列を処理する目的で設計された Perl,

本書で紹介した Ruby, 簡単な処理を手軽に書くためのオブジェクト指向言語 Python, 色々なハードウェアで安全に動作させる目的で設計されたオブジェクト指向言語 Java や C♯, web ページの中に簡単な処理を埋め込む目的で設計された JavaScript などがある。今日のソフトウェア開発で主に用いられるのはこれらの言語である。

10.2　プログラミング言語の種類

このようにプログラミング言語は沢山のものが作られているが、プログラムを作るときの考え方や、目的などによっていくつかの分類方法がある。

10.2.1　命令型言語と宣言型言語

Ruby もそうであるが多くの言語のプログラムは、「x と y を足して z に代入せよ」「n が正の間は以下の命令を繰り返せ」のように、コンピュータに対する命令を並べたものになっている。このような種類の言語は **命令型 (imperative) 言語**と呼ばれる。Ruby 以外でも、前出の中では C, C++, Java などは命令型言語に分類される。これらの言語は、機械語プログラムを人間が書きやすくなるように進化させていったもだとも言える。共通するのは、変数に値を代入することができる、条件分岐や繰り返し命令を使って命令の順序を制御する、関数やメソッド によって一連の命令をひとまとめにするといった機能を使ってプログラムを作る点である。

しばしば手続き型 (procedural) 言語とも呼ばれる。

命令型言語に対比されるのは**宣言型 (declarative) 言語**である。宣言型言語のプログラムは、物事の性質や関係を書き表わしたものであり、コンピュータはそれを基にして答えを探し出すことが計算が行われる。このよう言語は解き方が明らかでない問題を試行錯誤しながら解く場合などに多く用いられている。また、プログラムとしては性質や関係だけを書くことになるので、多くの場合には命令型言語よりも簡潔にプログラムを書くことができる。

以下、いくつかの宣言型言語のプログラムの例を見てみよう。

関数型言語

関数型言語は、数学の関数を使って関係を表わすような宣言型言語である。

以下に Haskell という関数型言語を用いて練習 4.13 (p.60) で紹介した Eratosthenes の篩を定義した例を示す。

```
1 primes = sieve [2..]
2  where sieve (p:xs)
3          = p : sieve [ x | x <- xs, x `mod` p > 0 ]
```

この例は次のような関係を記述している。

- 素数の列 (primes とは、2 から始まる数列 ([2..] を篩 (sieve) にかけたものである。
- 数列を篩にかけるとは、数列の先頭の値 (p) に「先頭以外の数列 xs のうち、p で割り切れないものを残した列」をつなげた (:) ものである。数学である条件を満たす値の集合を定義するときに $\{x \mid x \in \mathtt{xs} \ \wedge \ x \ \% \ \mathtt{p} > 0\}$ のような書き方をするが、[ ] の部分がこれに似せてあることが分かるだろうか。

この定義では列が持つべき性質だけを記述しており、どのような順序で値を求めるかについては書いていないことに注意してほしい。宣言型言語は、このような定義から適切な順序で計算を進める機能を持っている。

練習 10.1 (関数型言語のプログラム) 上に示したプログラムを primes.hs という名前のファイルに保存せよ。さらに、以下のようにして「ターミナル」から Haskell 言語の処理系を実行し、素数列が求められていることを確めよ。

```
1 cm12345$ hugs                    Haskell言語を実行する
2 Hugs> :load primes.hs            primes.hsを読み込む
3 Main> take 10 primes             先頭から10個取り出す
4 [2,3,5,7,11,13,17,19,23,29]
5 Main> primes !! 100              100番目を取り出す
6 547
7 Main> :quit                      Haskell言語を終了する
8 cm12345$ ▉
```

練習 10.2 (関数型言語による整列) 以下は Haskell によって書かれた整列プログラムである。

```
1 qsort [] = []
2 qsort (a:xs) = qsort [x | x <- xs, x <= a ]
3                ++ [a] ++ qsort[ x | x <- xs, a < x ]
```

このプログラムは、1 行目が「空の列 [] は整列されている」ことを表わしている。2 行目から 3 行目は、先頭が a, 残りが xs であるような列を整列すると、xs の中で a 以下の値を残した列を整列したものと、a と xs の中で a より大きい値を残した列を整列したものを順につなげた (++) ものであることを示している。

練習 10.1 を参考にして、qsort [3,1,4,1,5,9,2] などのような式を計算すると整列された列が求められることを確認せよ。

論理型言語

論理型言語は、論理式を使って関係を表わすような宣言型言語である。以下には Prolog という論理型言語を用いたプログラムを示す。

```
1  mother(kishiyoko, abeshinzo).      %岸洋子は安部晋三の母
2  father(abeshintaro, abeshinzo).    %安部晋太郎は安部晋三の父
3  father(abeshintaro, kishinobuo).  %安部晋太郎は岸信夫の父
4  father(kishinobusuke, kishiyoko).%岸信介は岸洋子の父
5
6  parent(X,Y) :- father(X,Y).      %X が Y の父ならば X は Y の親
7  parent(X,Y) :- mother(X,Y).      %X が Y の母ならば X は Y の親
8  sibling(X,Y)                     %Z が X の親で Z が Y の親ならば
9    :- parent(Z,X), parent(Z,Y). %X は Y の兄弟
10 grandparent(X,Y)                 %X が Z の親で Z が Y の親ならば
11   :- parent(X,Z), parent(Z,Y). %X は Y の祖父母
```

このプログラムは「岸洋子は安部晋三の母である」のような事実は`mother(kishiyoko, abeshinzo)` という形で表わし、「Z が X の親でかつ、Z が Y の親であるならば、X と Y は兄弟である」という規則を `sibling(X,Y) :- parent(Z,X), parent(Z,Y).` という形で表わす。このような関係だけを書き表わしておくと、あとは自動的に「安部晋太郎と岸信夫は兄弟か?」や「安部晋太郎の祖父母は誰か?」といった質問の答えを自動的に探してくれる。

下は上の定義を読み込んだ上で答えを探す様子である。下線部が入力で、`Yes`, `No` や変数 `X` にあてはまる値などが答として表示されている。

```
1  ?- sibling(abeshinzo,kishinobuo). %兄弟かどうか調べる
2  Yes
3  ?- grandparent(X,abeshinzo).      %祖父母を探す
4  X = kishinobusuke ;               %答が表示。さらに探す
5  No                                %もういない
```

10.2.2　オブジェクト指向言語

第 8 章で紹介したように、「もの」を中心にプログラムを作ってゆく種類の言語として**オブジェクト指向言語**がある。特徴的な機能には、データとそれに対する処理 (メソッド) をまとめた単位である**クラス**定義、**継承**による定義の流用などがある。

プログラミング言語が命令型か宣言型であるかという分類と、オブジェクト指向であるかどうは直交する性質のものなので、例えばオブジェクト指向かつ論理型であるようなプログラミング言語もある。

オブジェクト指向言語は、今日のソフトウェア開発で中心的に用られている。その理由の 1 つは、オブジェクト指向言語には、色々なプログラムで使われる典型的な定義をまとめた**ライブラリ**が充実しているためである。例えばウンドウ、メニュー、ボタンなどを用いたグラフィカルユーザインターフェース (GUI) を持ったプログラムを作る場合、ボタンやメニューなどの定義はほとんどのプログラムで共通している。オブジェクト指向言語では、これらの定義を**クラスライブラリ**として用意することで、容易にプログラムを完成させることができるようになる。

また、大規模な情報システムが扱う対象は必然的に複雑なるため、そのようなシステムを作る場合にもオブジェクト指向言語が用いられることが多い。人間が複雑な対象を整理するときは**階層的**に分類することが多くあるが、クラス定義の継承関係は、この階層性を直接表現できるのが、大きな理由の 1 つである。

10.2.3　スクリプト言語

スクリプト言語とは、簡単な処理を簡単に書けるように工夫された言語である。「スクリプト」という呼び方は、複雑な処理をするプログラム群を制御する台本 (script) のようなプログラムを書くために用いられることに由来している。実際、スクリプト言語の多くは、別のプログラムを起動したり文字列の処理をするための機能が用意されており、例えば「デジタルカメラから読み込んだ画像を、画像処理ソフトウェアを使って縮小して、電子メールソフトウェアを使って指定されたメールアドレスに送信する」のような処理を簡単に書くことができたりする。

スクリプト言語は、インタプリタ実行形式 (後述) のものが多い。これは、同じプログラムが異なるハードウェアでも動作できるようにする**互換性**のためと、プログラムを作って即座に実行できるようにするためといった理由による。しかしインタプリタ実行のために実行速度は、他のコンパイル実行形式 (後述) の言語と比べるとありま速くない。

10.3　プログラムの実行のされ方

コンピュータのハードウェアが直接実行できるのは機械語命令で書かれたプログラムだけであるため、高級言語のプログラムは間接的に実行されることになる。この実行のされ方は、大きく 2 つに分けられる。

1 つは**コンパイル実行**というもので、本格的なソフトウェアはこの方法で実行されるものが多い。もう 1 つは**インタプリタ実行**というもので、プログラムを作ったその場で実行してみるような場面ではこの方法が採られること

が多い。Ruby の処理系はインタプリタ実行に分類される。さらにまた、この 2 つの方法を組み合せたような方法もある。

10.3.1　コンパイル実行方式

コンパイル実行方式では、まず最初に高級言語で書かれたプログラム全体を機械語プログラムに翻訳する。この翻訳のことを**コンパイル**、翻訳をするプログラムのことを**コンパイラ**と言う。そして、翻訳された機械語プログラムを実行すると、ハードウェアは中に書かれている命令を 1 つずつ実行してゆき、元のプログラムが表わしているような計算を行う。

この方式は、次に説明するインタプリタ実行方式に比べると実行が速い。一方、翻訳という作業が入るために、プログラムを作ってから実行するまでの手間がかかる。また、プログラムの動きに異常があった場合、翻訳されたプログラムの動きから元の高級言語のプログラムの誤りを見つけなければいけないため、誤りを発見することが難しくなっている。そのため、この方式は一度プログラムを作って何度も実行するような場合に用いられる。

この方式で翻訳された機械語プログラムは、異なる種類の CPU では実行できない。従って例えば Cell Broadband Engine と呼ばれる種類の CPU で実行するために翻訳された機械語プログラムを、Intel 社の x86 と呼ばれる種類の CPU を持ったパーソナルコンピュータに読み込ませても実行はできず、高級言語で書かれたプログラムを翻訳し直す必要がある。

実行速度が重要なソフトウェア開発に用いられるプログラミング言語処理系の多くは、コンパイル実行方式をとっている。代表的なものとしては FORTRAN, C, C++ などはこの方式を取っている。

10.3.2　インタプリタ実行方式

インタプリタ実行方式では、CPU は**インタプリタ**と呼ばれる機械語プログラムを実行する。インタプリタは、高級言語プログラムの命令を 1 つずつ読み込み、それに応じた処理を行う。

例えば `1+1` という数式であれば、インタプリタプログラムは

1. 1 文字目「`1`」を見ると数字なので、それを数値 1 に変換し、変数 a にしまう
2. 2 文字目「`+`」を見ると足し算記号なので、それを変数 o にしまう
3. 3 文字目「`1`」を見ると数字なので、それを数値 1 に変換し、変数 b にしまう
4. ここで式の終わりに達したので、変数 o を見ると、+ なので、変数 a と b の値を加え、それを答えとする

というような計算をする。

インタプリタ実行方式はコンパイル実行方式よりも実行速度が遅い。その理由はこのように単純な計算であっても、式や演算子の種類が何であるかを 1 つ 1 つ判断しながら計算を進めてゆくためである。一方、コンパイル作業が不要なので、プログラムを即座に実行することができ、プログラムを簡単に作って実行したい場合に適している。

また、CPU の種類ごとにインタプリタを用意しておけば、同じプログラムを色々なコンピュータで実行することができる。

インタプリタ実行方式は、学習用の処理系やスクリプト言語処理系などで用いられている。

10.3.3　仮想機械による実行方式 *

コンパイル実行とインタプリタ実行を組み合わせた実行方式もある。その 1 つは**仮想機械**実行方式とよばれるものである。この方式では、架空のコンピュータである「仮想機械」を設計しておき、

- 高級言語のプログラムを仮想的機械の機械語プログラムに翻訳する。
- CPU は「仮想機械」というプログラムを実行する。このプログラムは、仮想的なコンピュータのシミュレータであり、仮想的機械の機械語プログラムを読み込んで実行することができる。

という方法でプログラムを実行するものである。

この方式は 2 つの実行方式の利点を兼ね備えている。つまり、

- インタプリタ方式と同じように、CPU の種類ごとに仮想機械を用意しておけば同じ(仮想機械)機械語プログラムを色々なコンピュータで動かすことができる
- インタプリタ方式より実行速度が速く、必要なメモリの量も小さい

という特徴がある。

仮想機械による実行方式を用いているプログラミング言語処理系には Pascal, Smalltalk, Java などがある。特に最近では仮想機械の作り方を工夫によって、コンパイル実行方式に遜色のない実行速度が達成されてきている。

10.4　C プログラムの紹介

Ruby 以外のプログラミング言語として **C 言語**のプログラムと、その実行方法を紹介しておこう。C 言語は 1970 年代初頭に UNIX オペレーティングシステムとそのアプリケーションソフトウェアを開発するために設計された

命令型言語である。C 言語の特徴は、高級言語であるにも関らずハードウェアを直接操作することができる点である。それでいて高級言語であるため、色々な種類の CPU で実行できる機械語プログラムを得ることができた。C 言語が発明された当時、オペレーティングシステムのプログラムの多くの部分くハードウェアを制御する必要があるためにアセンブリ言語によって作られていたことを考えると画期的な特徴だった。

10.4.1　C プログラムの特徴

C プログラムの特徴を見るために、第 6.1.3 節で紹介した Monte Carlo 法による円周率の計算を行うプログラムを C 言語で書き直したものを見てみよう (ファイル 10.1)。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  double montecarlo(int n) {/*def montecarlo(n)    */
5    int m, i;
6    double x, y;
7    m = 0;                    /*  m=0               */
8    for (i=1; i<=n; i++) {    /*  for i in 1..n     */
9      x = drand48();          /*    x = rand()      */
10     y = drand48();          /*    y = rand()      */
11     if (x*x + y*y < 1.0) {/*    if x*x + y*y <1.0*/
12       m = m + 1;            /*      m = m + 1     */
13     }                       /*    end             */
14   }                         /*  end               */
15   return m*1.0/n;           /*  m*1.0/n           */
16 }                           /*end                 */
17
18 int main(int argc, char** argv) {
19   srand48(time(NULL));
20   printf("%f\n", 4*montecarlo(1000000));
21 }
```

ファイル 10.1: mc.c

多くのプログラミング言語には、複雑な処理を行うための関数やデータの定義を集めた**ライブラリ**がある。C 言語にも色々なライブラリが用意されている。実際、文字例の表示や数学関数などは全てライブラリとして用意され

ている。1 行目、2 行目の#include で始まる行は、文字例の表示や乱数を使うための標準的なライブラリを使うための準備である。(Ruby でも数学関数を使うためには include(Math) を実行しなければいけないかったが、それと同じである。)

4 行目から 16 行目までが Monte Carlo 法によって四分円の面積を求めている関数である。右側に並べてあるのは Ruby による定義 (p.93 のファイル 6.2 に示したもの) である。(C 言語では/* */という記号に狭まれた部分が注釈として扱われる。)

C と Ruby の定義を見比べると、細かな点を除けばかなりの部分が似ている一方で、C の 4 行目の定義に double, int という語が加っていたり、5 行目、5 行目のように Ruby にはない命令がある。これらは**型に関する宣言**であり、変数にどのような種類の値をしまうか、関数がどのような種類の値を受け取り、返すかを表わしている。例えば 5 行目は変数 m, i には整数 (int) をしまうと宣言している。

型に関する宣言は何に用いられるのだろうか。第 6.3.1 節 (p.96) で見たように、実数と整数ではメモリ上の表現が異なり、その大きさも 32 ビットであったり 64 ビットであったりと一通りではない。また、足し算や掛け算のような演算は機械語命令では「整数の足し算」と「実数の足し算」で別の命令が用意されている。コンパイル実行方式の場合、プログラムを実行する前に変数をしまう場所を決めたり、計算式を機械語命令の並びに翻訳することになるが、その際に変数の型を用いて大きさや命令を選択している。例えば変数 m は 32 ビット整数 (int) と宣言されているので、翻訳された機械語プログラムの中では 32 ビット長のレジスタによって表わされ、m+1 という式は整数の足し算命令によって計算されるといった具合である。

レジスタとは CPU の中にあり、一時的に用いられるデータをしまうための回路である。

18 行目からの関数 main は、プログラムを実行したときに実行される関数である。ここでは乱数を使うための準備をして、montecarlo の計算を表示している。

10.4.2　C プログラムの実行

C プログラムは、コンパイル実行するのが一般的である。ここでは、Mac OS X や Unix OS 上で GCC という C 言語のコンパイラを用いて C プログラムを実行する手順を紹介する。

プログラムを作る: エディタを使ってプログラムを作成する。ここではファイル 10.1 の内容を mc.c という名前のファイルに作ることにする。

コンパイルする: 「ターミナル」で次のように入力する。

```
1 cm12345$ gcc -O mc.c
2 cm12345$
```

もし入力に間違いがあった場合には、エラーメッセージが行番号とともに表示される。その場合は、エディタに戻って誤りを修正する。

何も表示されない場合は機械語のプログラムが a.out という名前のファイルの中に作られている。

実行する: 続けて「ターミナル」で次のように入力する。

```
1 cm12345$ ./a.out
2 3.140104
3 cm12345$
```

練習 10.3 (C プログラムの修正と実行)　a) プログラムを実行する際に./a.out と入力するかわりに time ./a.out と入力すると、実行に要した時間を測定することができる。ファイル 10.1 に示したプログラムをコンパイルして実行し、実行に要した時間を測定せよ。

b) ファイル 10.1 に示したプログラムは 100 万回の繰り返しを行っているが、1000 万回行う場合の時間を測定せよ。

10.4.3　コンパイル実行方式と誤り

インタプリタ実行方式と比べると、コンパイル実行方式はプログラムを作成してから実行するまでの手間が多い。その一方で、コンパイルの際に誤りを見つけることができるので、場合によってはプログラムを実行しないでも間違いを修正することもできる。

どのような誤りを見つけることができるかの例をいくつか見てみよう。まずファイル 10.1 の 7 行目を k = 0; と変更したファイル mc1.c を作り、それをコンパイルしてみよう。

```
1 cm12345$ gcc -O mc1.c
2 mc1.c: In function 'montecarlo':
3 mc1.c:7: error: 'k' undeclared (first use in this function)
4 mc1.c:7: error: (Each undeclared identifier is reported only once
5 mc1.c:7: error: for each function it appears in.)
```

コンパイラは 7 行目で宣言されていない (undeclared) 変数 k が使われているとして、誤りを報告する。

さらに、この行を m = "abc"; と変更したファイル mc2.c を作り、それをコンパイルしてみよう。

```
1 cm12345$ gcc -O mc2.c
2 mc2.c: In function 'montecarlo':
```

```
3 mc2.c:7: warning: assignment makes integer from
    pointer without a cast
```

コンパイラは (m への) 代入 (assingment) によって、文字列 (pointer) を整数に変換してしまうとして、誤りの可能性を報告する。

C 言語では文字列は pointer という型のデータとして表わされている。

このように C 言語のようなコンパイル実行方式を採る言語では、変数や関数の名前の間違いや、値の種類に関する間違いをプログラムを実行する前にある程度発見することができる。

練習 10.4 (インタプリタ実行方式での間違いの発見) Ruby で書かれた Monte Carlo 法による円周率計算プログラム (p.93 のファイル 6.2) に、上で示したような変更を加えた場合、どのような誤りがいつ報告されるかを調べよ。プログラムのどの行で誤りが起きたと報告されるか?

10.4.4　実行方式と速度

コンパイル実行方式は、高級言語のプログラムを無駄のない機械語プログラムへと翻訳してから実行をするため、インタプリタ実行方式よりも高速である。実際に速度の違いがどの程度あるかを比べてみよう。

まず C 言語で書かれた Monte Carlo 法による円周率計算プログラム (ファイル 10.1) の実行時間を測ってみよう。

```
1 cm12345$ gcc -O mc.c
2 cm12345$ time ./a.out
3 3.143172
4
5 real    0m0.257s
6 user    0m0.244s
7 sys     0m0.004s
```

最後の 3 行が実行時間を示しているが、real と書かれた行が実際にかかった時間である。

ただしコンピュータは複数のプログラムを交替で動かすことがある。そのような場合には、「実際の」時間全てをこの C プログラムの実行に使っていたとは限らないことに注意せよ。

次に Ruby での実行時間を測ってみよう。

```
1 cm12345$ irb
2 irb(main):001:0> load("./montecarlo.rb")
3 => true
4 irb(main):002:0> require("benchmark") #時間測定の準備
5 => true
6 irb(main):003:0> puts(Benchmark.measure{
7                     montecarlo(1000000) })
8   5.660000   1.500000   7.160000 (  7.166609)
```

```
9 => nil
```

表示された数値のうち ( ) 内のものが実際にかかった時間であり、C プログラムを測定したときの real で書かれた行に対応している。

つまり、このコンピュータでは、 C プログラムの方が 30 倍近く速かったことになる。もちろんハードウェア、コンパイラ、インタプリタ、動かすプログラムの性質などによってこの差は大きく変化するので、一般的に言えるのはコンパイル実行方式の方がインタプリタ実行方式よりも速いことが多いということだけである。

練習 10.5 (実行方式と速度の比較) 以下に示すのは再帰的アルゴリズムによって Fibonacci 数を計算するプログラム (p.17 に示したファイル 5.1) を C 言語で書いたものである。コンパイルして time ./a.out 35 のように ./a.out に続けて整数 k を書いて実行すると fib(k) を計算する時間を測ることができる。

実際に計算時間を測定して、Ruby の場合 (65 に示したファイル 5.1) との実行時間の違いを調べよ。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int fibr(int k) {        /* def fibr(k)        */
5    if (k==0 || k==1) {    /*   if k==0 || k==1 */
6      return 1;            /*     1             */
7    }
8    else {                 /*   else            */
9      return fibr(k-1) +   /*     fibr(k-1) +   */
10            fibr(k-2);    /*     fibr(k-2)     */
11   }                      /*   end             */
12 }                        /* end               */
13
14 int main(int argc, char** argv) {
15   printf("%d\n", fibr(atoi(argv[1])));
16 }
```

ファイル 10.2: fib.c

10.4.5　機械語プログラムへのコンパイル *

機械語プログラムがどのようなものかを見るために、C プログラムからコンパイルされたプログラムを解読してみよう。多くのコンパイラは高級言語のプログラムをコンパイルする際に、アセンブリ言語のプログラムを経由して機械語プログラムに翻訳をしている。アセンブリ言語プログラムの命令は機械語の命令と一対一で対応しているが、人間が読むことができるテキスト形式で表現されているので、これを見るとどのような機械語プログラムが作られているかが分かる。

まずは「ターミナル」で C プログラムをコンパイルする際に以下のように -S という指示を加えてコンパイラを実行してみよう。

```
cm12345$ gcc -O -S mc.c
```

すると mc.s というファイルができる。定義が長いので関数 montecarlo に対応する部分だけを抜き出したものが以下である。それぞれの命令についての解説はしないが、右側に C 言語プログラム中の式や命令の対応を書き加えてある。

```
montecarlo:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %edi
        pushl   %esi
        pushl   %ebx
        subl    $12, %esp
        movl    8(%ebp), %esi
        movl    $0, %edi
        testl   %esi, %esi
        jle     .L4
        movl    $0, %edi        m = 0;
        movl    $1, %ebx        i = 1;
.L5:
        call    drand48         x = drand48();
        fstpl   -24(%ebp)
        call    drand48         y = drand48();
        fldl    -24(%ebp)
        fmul    %st(0), %st     y*y
        fxch    %st(1)
        fmul    %st(0), %st     x*x
        faddp   %st, %st(1)     x*x + y*y
        fld1                    1.0
        fucompp                 compare x*x+y*y and 1.0
        fnstsw  %ax
        sahf
        jbe     .L6             jump L6 if ≤
        addl    $1, %edi        m = m + 1;
.L6:
        addl    $1, %ebx        i++
        cmpl    %ebx, %esi      compare i and n
        jge     .L5             jump L5 if ≥
```

```
33 .L4:
34         pushl   %edi            m
35         fildl   (%esp)
36         movl    %esi, (%esp)    n
37         fildl   (%esp)
38         fdivrp  %st, %st(1)     /
39         addl    $16, %esp
40         popl    %ebx
41         popl    %esi
42         popl    %edi
43         popl    %ebp
44         ret
```

これを見て分かるように、10 行少々だった C プログラムは、40 個くらいの機械語の命令に翻訳されている。これは C 言語では `x*x + y*y < 1.0` のような 1 行で書かれていた数式の計算は、実際には乗算、加算、比較などの命令の並びに翻訳されていたりするためである。

練習 10.6 (機械語プログラムの編集と実行) 上のようにして得られた `mc.s` はアセンブリ言語のプログラムであるので、これを編集すればプログラムの動きが変わるはずである。次のようにしてこのことを確認せよ。

a) `mc.s` の 27 行目の `jbe .L6` という命令は、「直前の `x*x + y*y` と `1.0` との比較によって、後者 `1.0` の方が小さいか等しい場合には `.L6` へ進め」という命令である。 エディタを使ってこの行を `ja .L6` という命令に変更せよ。この命令は「直前の比較の後者が大きいか等しい場合には `.L6` へ進め」という意味になる。

b) 「ターミナル」で次のように入力して、編集されたアセンブリ言語プログラムを機械語に変換せよ。ファイル名が `mc.c` ではなくて `mc.s` であることに注意せよ。

```
1 cm12345$ gcc mc.s
```

c) 機械語プログラム `a.out` を実行し、その結果が、ファイル 10.1 の 11 行目の不等号を反転させたプログラムと同じであることを確認せよ。

練習 10.7 (機械語プログラムの編集) Fibonacci 数を計算する C 言語プログラム `fib.c` をアセンブリ言語プログラムに変換した上で、$\mathrm{fib}(k-1)+\mathrm{fib}(k-2)$ を行う足し算を引き算に変更せよ。k の値を変えて $\mathrm{fib}(k)$ をさせ、変更結果を確認せよ。

なお、Intel x86 という種類の CPU では足し算は `addl`, 引き算は `subl` という命令になる。